INRIA
CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France

Tél. : (1) 39 63 55 11

Rapports de Recherche

N°460

TECHNIQUES DE BASE SUR L'EXPLOITATION AUTOMATIQUE DU PARALLÉLISME DANS LES PROGRAMMES

Alain LICHNEWSKY
François THOMASSET

Décembre 1985

Techniques de Base sur l'Exploitation Automatique du Parallélisme dans les Programmes

Alain LICHNEWSKY
François THOMASSET

I.N.R.I.A.
Domaine de Voluceau
78153 Le Chesnay CEDEX

et

Université de Paris-Sud
91405 Orsay

RESUME

Ce rapport présente les techniques de base de la vectorisation telles qu'implémentées par les auteurs. L'intérêt de cette étude a été de montrer qu'il est possible de déduire complètement la théorie classique due à D.J.Kuck, L.Lamport, K.Kennedy, de modèles de parallélisme. Ceci permet d'envisager systématiquement l'approche d'architectures diverses. D'autre part les conditions de monotonie et d'approximation couvrent les besoins des réalisations pratiques. La réalisation "VATIL" qui est décrite ici sert actuellement à l'étude de l'impact de formes particulières de parallélisme, et des compromis approximation/ efficacité en vue de l'implémentation industrielle chez les constructeurs nationaux.

ABSTRACT

This paper reports on basic vectorization vectorization techniques as implemented by the authors in a prototype vectorizer. This study shows that one can deduce entirely the classical theory due to D.J.Kuck, K.Kennedy, L.Lamport, from models of parallel execution. Thus it is possible to consider systematically various computer architectures. On the other hand we establish monotony and approximation properties which are useful for practical purposes. The prototype "VATIL" which is described here is presently used for studying the impact of specific forms of parallelism, and of compromises between efficiency and approximations, in view of industrial implementations.

22 Novembre 1985

Techniques de Base sur l'Exploitation Automatique du Parallélisme dans les Programmes

Alain LICHNEWSKY
François THOMASSET

I.N.R.I.A.
Domaine de Voluceau
78153 Le Chesnay CEDEX

et

Université de Paris-Sud
91405 Orsay

1. Introduction

Notre objectif est de préciser le type de parallélisme qui peut etre mis en évidence dans les programmes avec comme points de départ une description algorithmique en FORTRAN usuel et une architecture Vectorielle générique. L'essentiel des concepts et techniques est actuellement bien connu, et largement du à D.J.Kuck, L.Lamport et K.Kennedy. L'exposition que nous en faisons ici vise essentiellement à en unifier le cadre, et *à faciliter l'application à une architecture donnée.* On tente de parvenir à cet objectif en formalisant des notions de programme parallèle adaptées. Ces concepts sont mis en oeuvre dans le prototype de vectoriseur multi-cible "VATIL", dont les résultats servent d'exemples dans cet article. (Cf. A.Lichnewsky et F.Thomasset [LT-a-paraitre]). Nous poursuivons actuellement nos développements d'une part dans la direction des processeurs MIMD-Vectoriels (Cray-XMP/ CDC-ETA-GF10), d'autre part vers l'optimisation de boucles multiples.

Les techniques que nous voulons illustrer ici peuvent etre décrites dans le cas d'un programme à un seul niveau ne faisant pas d'entrées- sorties. Ceci nous amene à définir brièvement la sémantique de l'exécution de ce programme suivant les formes de parallélisme mises en oeuvre. Notre référence est ainsi l'exécution séquentielle en FORTRAN usuel, et nous cherchons à caractériser l'équivalence entre les formes parallèles à promouvoir et la définition séquentielle de référence. La construction de versions parallèles à partir du programme source étant la "transformation" à laquelle nous souhaitons parvenir.

Suite à cette première caractérisation, nous serons en possession de critères permettant d'affirmer que l'exécution de boucles simples est possible en mode vectoriel. Leur application pure et simple permet une vectorisation dans certains cas particulièrement simples et conduit à un abandon global de la tentative de parallélisation dans des situations où il est souhaitable de proceder à une analyse plus fine voire à une transformation du programme. Pour cela nous commencons par étudier les obstructions à la parallélisation, qui sont des relations de dépendance entre instructions.

Les transformations qui sont ensuite proposées permettent de conserver des programmes équivalents mais réduisent les dépendances inhibant la parallélisation, ou bien permettent de localiser ou minimiser les portions non-parallélisables. Elles consistent essentiellement en l'introduction de variables supplémentaires, le reordonnancement d'instructions. La limitation

fondamentale que l'on trouve alors est constitué de dépendances circulaires qui constituent des récurrences. Il conviendra tout d'abord de minimiser la portion du programme affectée, par souci de performance, puis de tenter de trouver des méthodes permettant leur calcul tout en exhibant du parallélisme. Nous ne procéderons pas à une étude de ce dernier point sur lequel existe une importante littérature, en particulier lorsque il devient possible de se ramener à des opérations d'algèbre linéaire matricielle.

Une fois parvenu à ce point, il sera intéressant d'y ramener plusieurs situations a priori plus complexes faisant intervenir en particulier des tests conditionnels. Ceci sera possible sans introduire de concepts supplémentaires. Ce faisant, nous introduirons naturellement des récurrences booléennes, qui constitueront les obstructions les plus fondamentales du procédé.

Nous avons choisi de porter l'accent sur les applications des résultats que nous venons de passer en revue à des architectures concrètes auxquelles le lecteur pourra etre confronté. Ceci explique que nous passions ensuite directement à la prise en compte par nos outils de subtilités que l'on rencontre sur les architectures Cray-XMP et Fujitsu VP-200. Le lecteur trouvera à les appliquer dans d'autres situations a propos des matériels de nouvelle génération qui ne manqueront pas d'apparaitre. Plus précisément, nous évoquerons les problèmes liés à l'existence d'activités pipelines asynchrones sur lesquelles le processeur n'effectue de controle de cohérence sémantique que suite à l'exécution d'instructions explicites. Il aura bien fallu en reconnaitre la nécessité, et il nous semble qu'ici le problème ce poserait en les memes termes si nous étions partis d'un programme intrinsèquement parallèle écrit en FORTRAN-8X ou en ACTUS.

Un autre domaine qu'il nous sera possible d'évoquer est celui de l'utilisation de processeurs MIMD pour l'execution répartie de boucles de programmes. Ceci est en fait possible de facon rentable pour des architectures à mémoire partagée lorsque le cout des synchronisations est faible devant le temps d'exécution. Des réalisations intéressantes sont proposées pour le Cray-XMP, et ce point est au centre des développements du projet Cedar [GKLS83].

2. Représentation de Programmes Vectoriels.

2.1. Un sous-ensemble de FORTRAN

Pour la commodité du lecteur, nous précisons dans la Figure 1 un cadre simplifié qui permettra de s'affranchir d'un certain nombre de difficultés inhérentes à la définition de FORTRAN tout en préservant les principales difficultés que nous cherchons à illustrer. De plus nous nous limiterons à la situation où la tentative de vectorisation s'appliquera à des boucles DO explicites et mises dans une forme canonique. Sauf indication du contraire, les notations FORTRAN ont leur signification usuelle.

Les formes d'exécution parallèles seront définies de facon similaire, ceci fera que nous n'utiliserons pas l'interprétation opérationnelle de l'exécution sur un processeur vectoriel particulier pour justifier nos développements. Le choix de FORTRAN est quelque peu arbitraire et lié au fait qu'il s'agit du principal langage de programmation actuel dans le domaine du calcul scientifique. Ceci doit aussi permettre de comparer cet exposé aux réalisations industrielles qui opèrent en général sur ce langage.

```
< Programme>     :=   (< Declarations>) < Bloc> END
< Declarations>  :=   (DIMENSION  | INTEGER | REAL
                          | LOGICAL) < Ids>  (,< Ids>)*
< Bloc>          :=   (< Instructions>)+
< Instructions>  :=   (< Assign>  | < Do>   | < If>. | < Cond-exit>)
< Assign>        :=   < Var>  =  < Expr>
< Do> .1         :=   DO < L>  < Index>  =  1, < Niter>
       .2                    < Bloc>
       .3                 < L>  CONTINUE
< If> .1         :=   IF (< Expr>) THEN
     .2                      < Bloc>
     .3                  ELSE
     .4                      < Bloc>
     .5                  ENDIF
< Cond-exit>     :=   IF (< Expr>) GO TO < L>
```

Fig. 1 Syntaxe de Programmes Séquentiels

Dans cette définition, < Var>, < Expr> ... ont la signification usuelle de nom de variable, d'expression,... Les restrictions que nous ferons peuvent maintenant se résumer à:

Hypothèse 1:

< Ids>:

Identificateurs de variables en général dimensionnées.Nous sommes dans la situation où les dimensions des variables sont connues et ou l'on peut garantir qu'elles ont des représentations disjointes.

< Expr>:

Les expressions ne sont formées que de variables, éventuellement dimensionnées et de constantes. Il n'y a pas d'appel de fonctions.

< DO>:

Les boucles DO sont standardisées sous cette forme, ou le pas d'indice est 1. Ceci deplace en particulier les discussions sur le sens d'évolution de la variable d'induction au niveau de l'analyse des expressions d'indice, qui doit etre faite de toutes facons de manière détaillée ainsi que nous le verrons plus tard. < Niter> est un entier ou une constante donnant le nombre d'itérations dans la boucle, le fait d'en connaitre la valeur permet une exploitation plus fine, mais n'est pas une hypothèse réaliste. < Index> est la variable d'induction explicite de la boucle. Cette variable peut etre utilisée dans le < Bloc> corps de la boucle, où elle désigne l'itération courante, mais ne peut y faire l'objet d'une assignation. (1) Le nombre d'instructions du < Bloc> est appelé la "longueur" de la boucle. Diverses méthodes permettent de determiner les autres variables d'inductions, variant linéairement en fonction de < Index> et de leur substituer cette fonction.

< Cond-Exit>:

Cette instruction permet de sortir immédiatement de la boucle DO courante. La question de la sortie de boucles imbriquées ou du branchement à une autre instruction que celle suivant la boucle courante ne sera pas traitée ici.

(1) Ceci n'est pas une limitation, les normes FORTRAN-77 et FORTRAN-8X prévoient que le compte d'itération soit determiné à l'entrée et placé dans une variable inaccessible au programme.

2.2. Un modèle d'exécution séquentielle

L'effet du programme que nous venons de définir sera une transformation de l'état de la machine séquentielle partant d'un état initial vers un état final. L'état est la représentation de l'ensemble des variables apparaissant dans le programme, ces variables étant disjointes en ce sens qu'aucune action sur l'une d'elle ne saurait affecter les autres. Nous considérerons comme équivalents deux programmes effectuant la meme transformation état initial → état final, lorsque les données initiales permettent l'exécution sans erreurs.

Bien qu'il soit possible de procéder plus formellement, il nous suffit de nous donner l'interprétation de référence de cette transformation comme résultant de l'exécution d'une suite finie de transformations élémentaires correspondant chacune à l'exécution d'une instruction. Cette suite est ordonnée, et il sera commode d'en désigner un élément par le multi-indice formé du désignateur de l'instruction suivi de la valeur des indices d'induction des boucles imbriquées dans l'ordre d'imbrication.

```
A          DIMENSION X(10), Y(12)
B          DO 1 I= 1,2
C            DO 2 J= 1,3
D              X(3*I+ J)= I+ J
E        2   CONTINUE
F            Y(I) = I
G        1 CONTINUE
H            END
```

Exemple 1

L'exécution de ce programme correspond à la suite ordonnée des occurrences d'instructions:

$$\text{Trans(A)} \equiv \text{B, C(1), D(1,1), E(1,1), D(1,2),...., F(2), G(2),H.}$$

Cette suite est dite suite des Transitions d'où sa notation. Il est commode de noter que l'ordre d'exécution correspond aussi à l'ordre lexicographique sur ces n-uples. La relation d'ordre sera notée $<$ et on dira simplement que A précède B si $A < B$. Cette relation d'ordre est totale lorsque l'on est dans le modèle d'exécution séquentielle. Cette suite définit aussi le prédecesseur et le successeur d'une occurrence, dont nous pourrons etre amené à faire usage. Nous écrirons que A précède directement B si B= Succ(A) sous la forme $A \lessdot B$. Il n'y a pas lieu d'élaborer plus le modèle de l'exécution d'une instruction: les variables utilisées sont prises dans l'état hérité du prédecesseur, une expression en est évaluée, puis l'état modifié en conséquence. Le successeur est alors déterminé. Concernant les états, nous désignerons de la meme facon l'état résultant de l'exécution d'une occurrence d'instruction que cette dernière. On a maintenant la suite: (pour l'Exemple 1)

$$< \text{Initial} > \text{, B, ..., F(2), G(2), H} = < \text{Final} >$$

2.3. Manipulation des données : le cas sans Tests.

Pour arriver à des conditions utilisables, nous introduisons maintenant des sous-ensembles de l'état qui décrivent pour chaque occurrence d'instruction I(j,k,..) les données qu'elle utilise in(I(j,k,,..)) et celles qu'elle modifie out(I(j,k,..). Ceci se fait en distinguant les cas et récursivement : Bien entendu, les constantes ne participent pas à ces ensembles, et in(Expr) est l'union des variables qui y apparaissent

$$in(< Assign >) = in(< Expr >) \quad (1)$$

$$out(< Assign >) = < Var > \quad (2)$$

De manière très fine, interessante lorsqu'on étudie une boucle:

$$in(<Do>.1) = <Niter> \tag{3}$$

$$out(<Do>.1) = <Index> \tag{4}$$

$$out(<Do>.3) = <Index> \tag{5}$$

Lorsqu'on s'interessera à des manipulations plus globales, nous pourrons utiliser des notions moins fines:

$$in(<Bloc>) = \bigcup_{+} in(<Instruction>) \tag{6}$$

$$out(<Bloc>) = \bigcup_{+} out(<Instruction>) \tag{7}$$

$$in(<Do>) = <Niter> \cup in(<Bloc>) \tag{8}$$

$$out(<Do>) = <Index> \cup out(<Bloc>) \tag{9}$$

Le cas des tests ne peut pas toujours etre traité de facon satisfaisante par une approximation telle que:

$$in(<If>) = in(<Expr>) \cup in(<Bloc>) \cup in(<Bloc>)$$

ceci explique que nous preferions revenir plus tard sur ce point, de facon a ne pas alourdir l'exposé.

Concernant un ensemble d'occurrences de la meme instruction, nous avons les règles suivantes:

$$in(I(i)) = \bigcup_{j,k,l} in(I(i,j,k,l)) \tag{10}$$

$$out(I(i)) = \bigcup_{j,k,l} out(I(i,j,k,l)) \tag{11}$$

De manière à fixer les idée, en reprenant l'Exemple 1 ci- dessus on aboutirait aux ensembles suivants:

$$\begin{aligned} in(B) &= \phi \\ in(C) &= in(C(1)) = in(C(2)) = \phi \\ in(F(1)) &= in(F(2)) = in(F) = I \\ out(D(1,2)) &= A(5) \\ out(D(1)) &= A(4) \cup A(5) \cup A(6) \end{aligned}$$

2.4. Notations

Il nous arrivera d'avoir à détailler l'état suivant la variable ou l'elément de tableau calculé ou modifié. Ceci sera rendu possible par les notations, dont le domaine d'application sera précisé plus loin:

< Occurrence> .< Var> [< Index>]

Désigne l'élement < Index> de < Var> dans l'état résultant de l'exécution de < Occurrence> .

< Occurrence> .[< Index>]

Désigne l'élément < Index> de la variable out(< Occurrence>) dans l'état résultant de l'exécution de < Occurrence> .

< Occurrence> .Val[< Index>]

Désigne l'élément < Index> du résultat de l'évaluation du second membre de < Occurrence> .

< Occurrence> .< Index-Expr>

Désigne l'expression d'indice de la variable faisant objet de l'assignation, évaluée dans l'état pred(< Occurrence>).

< Occurrence> .< Index-Expr> [< Index>]

Désigne l'élément < Index> de l'expression d'indice de la variable vectorielle faisant objet de l'assignation vectorielle, ([2]) évaluée dans l'état pred(< Occurrence>).

Pour alléger les notations nous écrirons des expressions de la forme $A(\sigma).[I]$ sous la forme $A(\sigma,[I])$. D'autre part, nous noterons (1,,N) une indexation avec accès vectoriel, alors que [1:N] désignera un accès simultané.

Exemples:

Dans le contexte du fragment de programme:

```
          DO 1 I = 1, N
A::           X ( I*2 - 1 ) = Y (I) + 2
    1     CONTINUE
```

A(2).< Var>

désigne la variable X(3)

A.< Index-Expr>

désigne l'expression: I * 2 - 1

(A.< Index-Expr>)(2).Val

désigne la valeur 3

Dans le contexte de l'exécution simultanée (définie paragraphe 3.2):

```
          DO-SIM 1 I ∈ (1,,N)
B::           X ( I*2 - 1 ) = Y (I) + 2
    1     CONTINUE
```

B.< Var>

désigne le sous-tableau de X: X(1),X(3),X(5),...

B.< Var> [2]

désigne la variable: X(3) . Lorsque celà ne présentera pas d'ambiguité, nous utiliserons le synonyme: B.[2] .

B.Val[2]

([2]) ou simultanée ou concurrente (Cf. paragraphe 3)

désigne la valeur de: Y(2) + 2 .

B.< Index-Expr> [2]

désigne la valeur 3 .

3. Modèles d'exécution parallèle.

Nous donnons maintenant la syntaxe, la sémantique et des conditions de validité de plusieurs constructions représentant les formes de parallélisme sur lesquelles nous serons amenés à travailler. Les formes introduites sont multiples, et correspondent soit à la nécessité de décrire des architectures existantes en détail, soit à des formes intermédiaires qui seront utiles au niveau des raisonnements. Dans le meme type de contexte, on trouvera une méthode d'exposition analogue dans Lamport [Lam74,Lam76].

3.1. Exécution Concurrente.

Ceci sera spécifié par une instruction < Do-Conc> :

< Do-Conc> := DO-CONC < L> < Index> ∈ (1,..,< Nocc>)
< Bloc>
< L> CONTINUE

Figure 2

Ceci spécifie que lorsque pred(< Do-Conc>) est exécutée (i.e. dans l'état correspondant), les < Nocc> programmes < Bloc> (Index) séquentiels s'exécutent concurremment sans synchronisation aucune. Il n'est pas a priori possible de distinguer dans cette exécution globale une suite de transitions d'états telle que décrite ci-dessus. Cependant, si un seul des blocs était actif pendant un laps de temps, son exécution se ferait exactement selon le schéma séquentiel décrit plus haut.

Bien entendu, pour que ceci ait un sens, nous devons imposer une restriction, ce qui permet en définitive d'aboutir à la définition de l'exécution concurrente:

Définition 1:

Supposons que < Bloc> soit une séquence de programme pouvant utiliser mais ne modifiant pas la variable < Index> . L'exécution concurrente de < Nocc> occurrences est spécifiée par la construction < Do-Conc> , chaque occurrence recevant une valeur distincte de < Index> ∈ (1,2..,). Elle est définie lorsque :

$i \neq j \Longrightarrow out(<Bloc>_i) \cap out(<Bloc>_j) = \phi$ i)

$\Longrightarrow out(<Bloc>_i) \cap in(<Bloc>_j) = \phi$ ii)

Son effet est alors le meme que l'exécution séquentielle de la boucle,la valeur de l'index variant de 1 à < Nocc> .

Note:

La dernière phrase est volontairement imprécise, elle signifie que l'on obtient un programme équivalent en remplacant < Do-Conc> par le < Do> défini plus haut. Si l'on suppose que le < Do-Conc> est situé à l'intérieur d'un programme séquentiel (i.e. pas d'imbrication de constructions parallèles), on peut etre plus précis: la transformation Etat(pred(< Do-Conc>)) -> Etat(< Do-Conc>) est la meme que pour la boucle séquentielle.

Remarque:

Ceci est quelque peu différent des points de vue adoptés dans le cas usuel de l'étude de programmes parallèles. (Cf. D.Gries [Gri76], R.Milner[Mil80]). Dans notre cas, seule la signification du "calcul proprement dit", par opposition au controle, doit etre préservée: nos hypothèses en font une "boite noire" dont on ne connait que l'utilisation faite de la

mémoire. Ceci convient bien à la définition de l'outil automatique que l'on vise. On complète cette approche par des manipulations préalables du programme séquentiel tenant compte des propriétés d'associativité et de commutativité des opérateurs arithmétiques. (Cf. par exemple D.Kuck et Y.Muraoka [KM74]).

Un type d'algorithme qui est exclu par cette définition est la relaxation chaotique. En fait l'étude de la méthode numérique, et aussi de sa réalisation, a pour but de montrer que le résultat du < Do-Conc> est convenable, bien que non nécessairement équivalent au résultat séquentiel.

3.2. Exécution Simultanée

Ce mode de parallélisme représente fidèlement le fonctionnement de machines dites SIMD telles que ILLIAC-IV, ICL-DAP,... Dans ce cas les divers processeurs executent pas à pas le meme programme, de facon totalement synchrone. En outre, formaliser cette situation va nous permettre d'aborder plus facilement le cas du mode vectoriel par la suite. Ceci sera spécifié par une instruction < Do-Sim>:

< Do-Sim> := DO-SIM < L> < Index> ∈ (1,..,< Nocc>)
< Bloc>
< L> CONTINUE

Figure 3

Ceci indique que les < Nocc> occurrences de < Bloc> s'exécutent suivant la meme séquence de pas élémentaires. Ceci est beaucoup plus rigide que le mode concurrent, ce qui va nous permettre de définir l'état final sous des conditions beaucoup moins restrictives que celles de la Définition 1. Toutefois cet état final ne sera pas nécessairement identique à celui produit par l'exécution séquentielle.

Nous devons tout d'abord imposer des conditions assurant la similarité des occurrences de < Bloc>:

Condition de Similarité (11).

Soit I la variable indexant le < Do-Sim> , et B le < Bloc> sur lequel il porte.

i) B est construit exclusivement a partir de < Assign> < Do> et < Bloc> . I n'est pas modifiée dans B.

ii) Les < Niter> des boucles < Do> incluses dans B sont constantes dans le < Do-Sim> : ne dépendent ni de I ni de $\bigcup_i out(B_i)$

Ceci étant, l'exécution de B passe par une suite d'états bien définie et indépendante de I: pred(< Do-Sim>), E_1, E_2,,, E_n. ([3]) La suite des états: Pred(< Do-Sim>), $E_1[1{:}N]$, $E_2[1{:}N]$,... résultant du < Do-Sim> s'en déduit: à l'étape $E_j[1{:}N] \rightarrow E_{j+1}[1{:}N]$ toutes les occurrences de $E_{j+1}[I]$ recoivent des valeurs distinctes i de l'indice I et prennent leurs autres données in($E_{j+1}[i]$) à l'état $E_j[1{:}N]$. L'état $E_{j+1}[1{:}N]$ résulte ensuite de l'assignation à out ($E_{j+1}[i]$) du résultat calculé par $E_{j+1}[i]$; ceci est bien défini car nous imposons la

Condition de non interférence (12):

i) Pour toute occurrence d'instruction de B:

$$i \neq j \Longrightarrow out(B_i) \cap out(B_j) = \phi$$

([3]) Ceci est une notation abrégée pour la suite des occurrences d'instructions définie plus haut.

On aboutit enfin à la

Définition 2:

Supposons les deux conditions ci-dessus vérifiées, l'exécution simultanée de < Nocc> occurrences de < Bloc> est spécifiée par la construction < Do-Sim> . Si la suite des états résultant de l'exécution séquentielle d'une occurrence de < Bloc> est :

$$(E_1(multi\text{-}indice_1), E_2(multi\text{-}indice_1), ...) \text{ que l'on note } (E_1(\sigma), E_2(\sigma), ...)$$

alors celle des états résultant du < Do-Sim> est :

$$(E_1(\sigma, [1{:}< Nocc >]), E_2(\sigma, [1{:}< Nocc >]), ...)$$

Propriété 1:

Soient B un < Bloc> et I une variable tels que A = (DO-SIM I Niter B) soit défini. Supposons

$$Niter \bigcap \bigcup_{i \in (1,,Niter)} out(B_i) = \phi$$

Alors A est équivalent au < Bloc> B' obtenu en concaténant le résultat de la transformation des instructions d'assignation de B: S_k par (DO-SIM I Niter S_k)

Preuve:

Il suffit d'examiner successivement les cas où B est l'un des 3 types d'instructions autorisées.

Remarques:

i) Lorsque B ci-dessus est une simple assignation, nécessairement vectorielle et indexée par I, A a la meme signification qu'une assignation de tableau mono-dimensionné dans les projets FORTRAN-8X ou VECTRAN (Cf. Example 2 below). Certaines des considérations ci-après sont particulièrement importantes comme la question du "Blocage" de telles boucles, et celle de la nécessité de recourir ou non à un temporaire lors de l'évaluation scalaire ou vectorielle. (Cf. [ANSI84], G.Paul [Pau82]).

```
                                C (avail (n . 50))
1   ;inst-101:U          >      DO 1 i = 1 ,n ,1
2   ;inst-102:S          >          x(i) = 2 * z(i) + 4
3   ;inst-103:S          >          y(i) = 1+x(i) * z(i)
4   ;inst-101:U          >1         CONTINUE
```

Example 2-a

```
1   ;inst-101:U          > C  Dovec 1 traduit en Fortran-8x :
2   ;inst-102:S          >         x(1:n:1) = 2 * z(*) + 4
3   ;inst-103:S          >         y(1:n:1) = 1 + x(1:n:1) * z(*)
4   ;inst-101:U          > C       Fin Dovec 1.
```

Exemple 2-b: Traduction en Fortran-8x.

ii) On note que A = (DO-SIM I Niter B) n'est équivalent ni a (DO I Niter B) ni au résultat de la transformation ci dessus ou le remplacement est fait par un DO séquentiel. On vérifiera ceci sur l'exemple:

```
      DO-SIM 1 I ∈ (1, ..., 10)
           X(I) = X(I-1) + X(I+1)
           Y(I) = X(I)
1     CONTINUE
```

Exemple 3

iii) (Blocage de boucles)

La vraie raison d'etre de la construction DO-SIM est l'exécution sur les machines SIMD. Dans ce cas, il y a en pratique une limitation supérieure à < Niter> , qui ne saurait dépasser un maximum physique. Pour traiter des boucles plus longues, on procède à leur "blocage". Alors que le blocage de boucles séquentielles est trivialement possible par la transformation:

```
      DO 1 I = 1, n*m
         < Bloc>
1     CONTINUE

      DO 1 I'= 1, m
         DO 2  I'' = 1, n
           I= I'' + n * (I'-1)
           < Bloc>
2        CONTINUE
1     CONTINUE
```

Exemple 4

Il est clair que l'on ne saurait en faire autant dans le cas de DO- SIM, sans une analyse plus complète par les méthodes que nous introduirons ci-après. Considérons en effet la transformation convenable dans le cas:

```
      DO SIM 1 I =(1,,1000)
         X(I)= X(1000-I)
1     CONTINUE
```

Boucle initiale

```
      DO 1 I'= 1, 10
         DO 2 SIM I'' = (1,, 100)
              I= I'' + 100 * (I'-1)
              X'(I) = X(1000-I)
2        CONTINUE
1     CONTINUE

      DO 3 I'= 1, 10
         DO 4 SIM I'' = (1,, 100)
           I= I'' + 100 * (I'-1)
           X(I) = X'(I)
4        CONTINUE
3     CONTINUE
```

Résultat de la transformation
Exemple 5

Elle nécessite l'introduction d'une variable intermédiaire, et la distribution de la boucle globale. Cette difficulté est aussi présente dans l'implémentation de FORTRAN-8X, d'autant plus qu'il ne saurait etre question d'utiliser systématiquement le procédé illustré ci-dessus sous peine d'une importante pénalisation, les super-calculateurs étant presque toujours limités par le débit de leur mémoire.

Remarque:

Nous venons de décrire l'exécution simultanée d'une famille à un paramètre de < Bloc> . Le cas de familles à plusieurs paramètres voire non structurées s'y ramène, en construisant une indexation convenable.

3.3. Exécution Vectorielle - 1

La première situation correspond à celle des machines type Cray-1S, CDC-Cyber-20X, qui sérialisent leurs interactions avec la mémoire. Ceci permet une description extrêmement proche du mode simultané ci-dessus, les interactions avec l'état que notre modèle utilise étant en fait séquentielles. La principale modification est que le mode vectoriel privilégie un ordre d'exécution par le biais d'une *mono-indexation des < Blocs> exécutés.* Ceci peut etre assorti de conditions sur une réalisation donnée (pas constant, pas de 1, accès indirect,...).

Remarque:

Certains processeurs décrits par ce modèle sont capables d'activités concurrentes, qu'ils synchronisent par analyse du flot de données. Dans le cas du Cray-1S, ceci est effectué par un mécanisme de réservation des registres, qui permet le chainage. Ceci ne pose pas de problème dans notre modèle, la cohérence sémantique étant préservée par la combinaison de deux mécanismes : l'analyse du flot de données dans les registres, la stricte sérialisation des opérations avec la mémoire.

Ceci sera spécifié par une instruction < Do-Vect> :

```
< Do-Vect>  :=  DO-VECT < L>  < Index>  =  1,..,< Longv>
                    < Bloc>
                < L>      CONTINUE
```

Figure 4

Ceci indique que < Bloc> est une suite d'assignations portant sur des vecteurs indexés par une expression de < Index> . On convient d'étendre tout scalaire utilisé dans une expression en un vecteur dont toutes les composantes sont identiques. ([4]) Le résultat est toujours indexé, l'assignation à un scalaire ne pouvant intervenir que dans le cas d'une indexation très dégénérée et il vaut mieux assigner l'expression correspondant à la dernière valeur de l'index, lorsque on en est conscient. Le cas des récurrences (réduction) non vectorielles est explicitement exclu.

Il convient maintenant de faire les hypothèses permettant de donner un sens à ceci, selon la meme démarche que ci-dessus. Toutefois, nous ne pouvons faire l'équivalent de l'hypothèse de "non-interférence" ci-dessus, ce qui rend l'écriture d'un bloc contenant des boucles DO peu naturelle et nous conduit *à nous limiter au cas d'un bloc constitué d'une suite d'assignations.* En effet, ici et sans cette hypothèse, on ne peut déduire l'équivalent de la propriété 1 ci-dessus.

Caractère Vectoriel.

Soit I la variable indexant le < Do-Vect> , et B le < Bloc> sur lequel il porte.

i) B est construit exclusivement à partir de < Assign> . I n'est pas modifiée dans B. Les variables et constantes scalaires pourront faire l'objet d'extensions vectorielles.

ii) Les variables faisant l'objet d'assignation sont indexées par une expression de I.

Note:

([4]) Ceci est en fait directement supporté au niveau architectural sans pénalité.

En pratique des conditions supplémentaires sont imposées aux expressions d'index. Les instructions d'accès aléatoire à la mémoire permettent de s'en affranchir si bien que nous n'entrons pas dans ce type de considérations ici.

Effet du < Do-Vect> (13)

i) Soit (B_k) la liste des instructions du bloc, l'effet du < Do-Vect> est la combinaison séquentielle des boucles vectorielles élémentaires ($< Do- VectI=1,,VL\ B_k >$).

ii) Si A est une assignation, (< Do-Vect> I = 1,,VL A) est défini sous la condition de **Non-Recurrence:** ([5])

$$1 \leq i < j \leq VL \implies out(A(i)) \cap in(A(j)) = \phi$$

Son effet est le meme que l'exécution séquentielle (< Do> I = 1,VL A)

On aboutit ainsi à la

Définition 3:

Supposons les deux conditions ci-dessus vérifiées respectivement par le < Bloc> et chacune de ses assignations prises isolément. L'exécution vectorielle de < Bloc> est définie comme étant la suite des exécutions vectorielles des assignations de < Bloc>. L'effet de l'exécution vectorielle d'une assignation produit le meme résultat que l'exécution de la boucle sequentielle correspondante.

Remarque:

La condition de non-récurrence ci-dessus suffit à définir l'effet du < Do-Vect> car nous avons supposé que les transferts vers la mémoire étaient sérialisés. Ceci est en particulier utilisé pour que l'effet de la suite d'assignations possibles à une meme variable fournisse la valeur de la dernière. Cette hypothèse est a vérifier au niveau de l'architecture du calculateur cible; si ce n'est pas le cas il convient d'ajouter à la condition de Non-Récurrence la condition supplémentaire :

$$\bigcap_{1 \leq i \leq VL} out(A(i)) = \phi$$

Ceci est inutile sur la majorité des calculateurs vectoriels actuels.

3.4. Exécution Vectorielle - 2

Il nous faut maintenant envisager le cas des machines vectorielles de seconde génération, capables en particulier d'activités asynchrones. Au niveau architectural, les possibilités sont nombreuses, et tournent autour du principe qu'il est possible d'analyser le flot de données portant sur un petit nombre de variables (registres), alors que l'analyse semblable des transferts avec la mémoire est hors de portée en pratique. Ceci est complété par des procédés de sérialisation adéquats qui permettent la prise en compte explicite des opérations sur la mémoire. Cette sérialisation doit etre effectuée par logiciel à bon escient, l'utilisation de sérialisations non-nécessaires se traduisant par une baisse des performances qui peut etre sensible.

Il n'est pas de notre propos de détailler ici ces mécanismes, bien au contraire de leur substituer un modèle abstrait. Etant donné leur relative nouveauté, nous en donnons quelques exemples dans la table ci-après. ([6])

([5]) Dans la formule ci-dessus, A(i) désigne l'instruction A exécutée dans l'état pred(< Do-Vect>) ou on a substitué à I la valeur i.

([6]) Le terme transfert désigne les opérations de lecture et écriture sur la mémoire.

Cray - XMP:

i) Transferts scalaires sérialisés entre eux et par rapport aux transferts vectoriels.

ii) Les lancements de Transferts vectoriels sont sérialisés mais deux lectures s'effectuent en meme temps qu'une écriture.

iii) Possibilité d'attendre la fin des activités lancées, par l'instruction CMR (Complete Memory Reference). Possibilité de sérialiser les lectures par rapport aux écritures (Mode Mono-/Bi-directionnel).

Fujitsu VP:

i) Sérialisation entre activités scalaires et vectoriels par instructions VWAIT, VPOST définissant une "fenètre".

ii) Deux canaux d'accès à la mémoire bidirectionnels. Chacun travaille séquentiellement et il est possible d'imposer le canal d'accès pour une opération sur la mémoire.

Ces modes d'exécution seront traités par un modèle unique, mettant en oeuvre concurrence et synchronisation par les données. Il sera spécifié par une instruction < Do-Cvect> :

```
< Do-Cvect>    :=   DO-CVECT < L>  < Index>  =  1,..,< Longv>
                       < Synch-List>
                       < Bloc>
                  < L>  CONTINUE
< Synch-List> :=      SYNCH < Var>   (, < Var> )*
```

Figure 5

Très informellement, ceci indique que la suite des assignations que comporte le < Do-Cvect> sera exécutée de facon concurrente, chaque assignation se vectorisant comme ci-dessus. Toutefois, une sérialisation est effectuée pour respecter le flot de données concernant les variables de la liste < Synch-List> . Le < Do-Vect> coincide ainsi avec un < Do-Cvect> où l'ensemble des variables figure dans la liste de synchronisation. Bien entendu un < Do-Cvect> inséré dans un bloc d'instructions ordinaires est exécuté en partant de l'état hérité de son prédécesseur et sera terminé fournissant l'état dans lequel son successeur s'exécutera.

Pour clarifier ces diverses particularités, donnons un exemple: la boucle séquentielle suivante :

```
      DO  1 I= 1, 1000
        AR = CX(5,I)
        BR = AR - PX(5,I)
        PX(5,I) = AR
        CR = BR - PX(6,I)
        PX(6,I) = BR
        PX(7,I) = CR
1     CONTINUE
```

Exemple 6.

peut etre transformée en une succession de deux boucles vectorielles mettant en oeuvre des unités pipelines asynchrones:

```
      DO-CVECT 1 I= 1,1000
      SYNCH AR', BR'
        AR'(I) = CX(5,I)
        BR'(I) = AR'(I) - PX(5,I)
        CR'(I) = BR'(I) - PX(6,I)
1     CONTINUE
      DO-CVECT 2 I= 1,1000
        PX(5,I) = AR'(I)
        PX(6,I) = BR'(I)
        PX(7,I) = CR'(I)
2     CONTINUE
        AR    = AR'(1000)
        BR    = BR'(1000)
        CR    = CR'(1000)
```

Exemple 7.

L'intérêt de la solution illustré sur cet exemple, tiré du jeu de test classique des "Boucles de Livermore", est que:

i) dans la première boucle, les variables sur lesquelles la synchronisation a lieu sont stockées dans des registres ce qui rend la synchronisation possible par l'analyse des dépendances entre registres implémentée au niveau matériel. Ceci permet le chainage et n'est pas pénalisant. Le fait que les variables AR',BR' et CR' soient obtenues par expansion d'un scalaire fait que l'on n'aura pas à les représenter en mémoire.

ii) dans la seconde boucle, les activités sont totalement asynchrones.

iii) il a fallu effectuer une seule synchronisation explicite, apparaissant dans la sérialisation de l'exécution des deux boucles, ce qui a suffi à éviter les conflits au sujet des opérandes PX, probablement stockés en mémoire. Ce dernier point n'a pu etre obtenu qu'en déplacant une instruction.

Synchronisation par les données (14) :

Soient $E_1, E_2, \ldots, E_k$ les k instructions du < bloc> . Dans l'état résultant de l'exécution de pred(< Do-Cvect>), les k boucles vectorielles (< Do-Vect I = 1,,VL E_i), s'exécutent concurremment. La restriction qui est apportée est que s'il existe une variable A $\in$ < Synch-list> , et deux indices i < j avec :

$$A \in \bigcup_{I=1,,VL} out(E_i(I)) \quad et \quad A \in \bigcup_{I=1,,VL} in(E_j(I)), \qquad a/$$

ou

$$A \in \bigcup_{I=1,,VL} in(E_i(I)) \quad et \quad A \in \bigcup_{I=1,,VL} out(E_j(I)), \qquad b/$$

ou

$$A \in \bigcup_{I=1,,VL} out(E_i(I)) \quad et \quad A \in \bigcup_{I=1,,VL} out(E_j(I)) \qquad c/$$

Alors les accès à la variable A par l'instruction E_i précèdent ceux faits par E_j.
Nous devons maintenant imposer une restriction, donnant un sens aux activités concurrentes. Elle est du meme type que celle s'appliquant au < Do-Conc> mais doit etre particularisée . Ceci permet d'aboutir à la:

Définition 5.

Supposons que < Bloc> soit une séquence de programme soumise aux memes restrictions que dans le cas du < Do-Vect> défini ci-dessus. L'exécution du < Do-Cvect> est définie lorsque:

$$i \neq j \Longrightarrow \bigcup_{I=1,,VL} out(E_i(I)) \cap \bigcup_{I=1,,VL} out(E_j(I)) \subset \bigcup_{A \in <Synch-List>} A \qquad (Def5.i)$$

$$\Longrightarrow \bigcup_{I=1,,VL} out(E_i(I)) \cap \bigcup_{I=1,,VL} in(E_j(I)) \subset \bigcup_{A \in <Synch-List>} A \qquad (Def5.ii)$$

Son effet est alors le meme que l'exécution de la boucle vectorielle (< Do-Vect> I = 1,,VL < Bloc>).

Note:

Nous ne venons pas de démontrer que les synchronisations spécifiées ci-dessus permettent de prouver le comportement de la boucle < Do-Cvect> . Ces synchronisations ont été explicitées pour la compréhension du modèle par le lecteur, notre modèle se trouvant défini "axiomatiquement". Le lecteur pourra cependant utiliser notre description détaillée pour s'assurer que le modèle de la définition convient pour représenter le comportement d'un matériel particulier.

Ceci est, en particulier, du au fait que nous souhaitons développer cet exposé indépendamment de tout modèle d'exécution concurrente.

3.5. Théorie de Kuck: Analyse du flot de données.

La méthode usuelle, employée en particulier par D.J.Kuck et son école est de partir d'un graphe de dépendance puis d'étudier l'effet de transformations de programmes sur ce graphe. De nombreuses adaptations ont été faites de cette technique, couvrant en particulier le cas des boucles emboitées. (Cf.D.J.Kuck [Kuc81], R.A.Towle [Tow76], J.R.Allen & K.Kennedy [AK82]). Ceci est extrèmement efficace du point de vue pratique, mais nous semble avoir l'inconvénient de masquer les différences parfois subtiles qui existent entre les diverses constructions parallèles. Nous allons donc montrer que l'on peut retrouver les memes notions en partant des définitions ci-dessus. Ceci est donc un effort dans le sens d'une dérivation de cette théorie à partir de propriétés sémantiques des constructions du programme parallèle. Elle devrait logiquement etre poursuivie par une justification formelle du formalisme retenu et de certaines démonstrations. Dans la pratique il est nécéssaire de s'assurer que la réalisation des constructions de base vérifie bien les propriétés ci-dessus.

La raison principale qui nous fait penser que cette approche est intéressante et complète les résultats disponibles est qu'elle permet d'entrevoir une adaptation facile à des formes de parallélisme voisines mais pouvant présenter des différences subtiles. Notre construction peut

dans ce cas etre reprise à partir des définitions et permettra d'aboutir systématiquement à une variante convenable de l'ensemble de techniques dues à Kuck. Dans le cas personnel des auteurs, c'est probablement la nécessité d'exposer l'adaptation de la technique pour les Cray-XMP et Fujitsu-VP200 qui a motivé ce développement.

3.5.1. Transformation d'un < Do> séquentiel en < Do-Sim> .

Nous cherchons donc une condition qui permette d'assurer que l'exécution simultanée d'un bloc est possible. Conformément aux hypothèses faites plus haut nous excluons provisoirement les instructions < If> et < Cond-Exit> . Nous partons donc de:

Inst-1 := (< Do> I = 1, N ($A_1,,,A_k$))

Nous espérons, que sous des conditions à préciser, ceci sera équivalent à:

Inst-2 := (< Do-Sim> $I\in(1,,N)$ ($A_1,,,A_k$)) ; I = N

Il nous faut tout d'abord imposer les conditions techniques sous les quelles Inst-2 est définie, et qui ne sont pas nécessaires dans le cas de Inst-1:

i) Les < Niters> des boucles incluses n'appartiennent pas à out(Inst-1).

ii) **(Condition de non-interférence) : (15)**

pour $1\leq i\leq k$ *et* $I\neq J$, $1\leq I,J\leq N$:

$$out(A_i(I)) \cap out(A_i(J)) = \phi$$

Maintenant il nous faut donner des conditions pour que Inst-1 et Inst 2 soient effectivement équivalentes ce qui n'est généralement pas vrai sous les seules conditions ci-dessus. Ceci est illustré par les deux exemples suivants, dont le premier est suggéré par la Propriété 1 ci dessus:

```
      DO 1  I = 1, 1000                    DO 1 I = 1, 1000
        X(I) = Y(I-1)          ≠             X(I)= Y(I-1)
        Y(I) = Z(I)                      1 CONTINUE
    1 CONTINUE                           DO 2 I = 1, 1000
                                           Y(I) = Z(I)
                                         2 CONTINUE

     DO 1  I = 1, 10000                    DO-SIM I = 1, 1000
       X(I) = X(I-1) + X(I+1)    ≠          X(I)= Y(I-1)
   1 CONTINUE                           2 CONTINUE
```

La seule base que nous puissions utiliser pour raisonner sur l'équivalence de Inst-1 et Inst-2 est l'existence des suites de transition d'état définies plus haut: ([6])

$$Trans(Inst-1)=<Initial>\overset{\sim}{<}<Inst-1>.1\overset{\sim}{<}A_1(1)<\cdots \qquad (16)$$

$$A_l(1,\sigma)<A_k(1)<<Inst-1>.3(1)....$$

$$<Final>$$

$$Trans(Inst-2)=<Initial>\overset{\sim}{<}<Inst-1>.1\overset{\sim}{<}A_1([1:N])<\cdots \qquad (17)$$

$$A_l(\sigma,[1:N])<A_k(1,[1:N])<\cdots<Inst-1>.3([1:N])....$$

([6]) Dans l'équation ci après, on a noté < Inst> .1 l'entete de l'instruction < Do> suivant la Figure 1.

où $A_l(1,..)$ désigne l'occurence de l'instruction A_l dans laquelle la variable d'induction I a la valeur 1, et dans laquelle les variables d'induction de toutes les boucles imbriquées dans A_l sont spécifiées mais simplement abrégées dans la formule par σ. ([7])

L'idée est maintenant de se ramener à la démonstration de Bernstein [Ber66] en considérant les assignations de valeur scalaires. La gestion des boucles éventuellement imbriquées doit tout d'abord etre examinée, pour voir qu'il y a correspondance entre les occurences des assignations complètement indicées. Ensuite on peut considérer dans la preuve, en ce qui concerne les boucles imbriquées:

$$< Do > .1 \approx I = 1 \ ; \ < Do > .3 \approx I = I + 1 \tag{18}$$

Dans Trans(Inst-1), chaque transition entre deux états consécutifs correspond à une telle assignation. Dans Trans(Inst-2), une instruction ne spécifie pas une suite séquentielle d'assignations scalaires. Cependant l'instruction $A_l(\sigma,[1{:}N])$ spécifie N assignations aux variables $A_l(1).< Var >$, $A_l(2).< Var >$,... Ceci nous permet de décomposer la suite Trans(Inst-2) en un ensemble d'assignations "élémentaires" non nécessairement sérialisables. Nous allons utiliser cette vue de la suite Trans(Inst-2) pour vérifier que chaque élément de tableau ([8]) recoit bien la meme valeur que dans le cas séquentiel. Nous noterons donc $A_l(\sigma,*[m])$ l'action de calculer $A_l(\sigma,\ [1{:}N]).<$ Var> (m) en prenant les données dans $Pred(A_l(\sigma,\ [1{:}N]))$; le résultat se note $A_l(\sigma,\ [m])$. Le résultat assigné est la valeur de l'élément de < Var> (m) dans l'état $A_l(\sigma,\ [1{:}N])$ résultant de l'exécution simultanée de l'assignation.

La bijection entre assignations séquentielles et résultat produit dans les itérations simultanées est l'objet du:

Lemme 1

Sous les hypothèses ci-dessus, il y a bijection entre la suite Trans(Inst-1) et l'ensemble des valeurs produites ([9]) par l'exécution de Inst-2 :

$$\bigcup_{l,\sigma,m} A_l(\sigma,*[m]) \tag{19}$$

Cette bijection, notée $\approx$ est définie par: ($\dot{\sigma}=\ddot{\sigma}$):

$$A_l(m,\dot{\sigma}) \approx A_l(\ddot{\sigma},*[m]) \tag{20}$$

Preuve:

Il suffit de remarquer que les valeurs limites < Niter> des boucles imbriquées dans < Inst-1> et < Inst-2> ne sont pas modifiées du fait de l'hypothèse ci-dessus. Dans le cas de la boucle < Inst-1> I prend consécutivement dans toutes les instructions $A_1, \ldots, A_k$ les valeurs 1 , puis 2..., puis N. Ceci provient de (Hyp. 1.< DO>). Dans le cas de la boucle < Inst-2>, les résultats sont indexés par $m \in [1{:}N]$.

([7]) On note que σ n'est qu'une abréviation et que deux occurrences de σ sont a priori distinctes. Lorsqu'on voudra désigner des valeurs et non un multi-indice générique on ecrira $\dot{\sigma},\ddot{\sigma}$ ...

([8]) Nous analysons effectivement les transformations simultanées avec une granularité plus faible, à ce niveau ce n'est pas une suite de transformations élémentaires consécutives .

([9]) Il s'agit ici d'un abus de langage, il serait plus exact de dire "productions de valeurs"; car, ainsi qu'il est d'usage dans l'analyse du flot des données, on ne tient pas compte de l'égalité éventuelle entre deux valeurs produites. Ici les valeurs produites sont indexées par le produit des occurrences d'instructions et de la longueur des vecteurs (ou nombre d'évaluations simultanées), d'où la formule. Ceci correspond au nombre de seconds membres évalués, mais non nécessairement au nombre d'éléments de l'état affectés car une instruction vectorielle peut affecter plusieurs valeurs au meme élément de tableau.

Pour montrer l'équivalence de < Inst-1> et < Inst-2> , nous allons en fait montrer que pour chaque elément de out(< Inst-1>) = out(< Inst-2>) qui ne soit pas la variable d'induction I, la suite des valeurs prises est identique dans les deux exécutions. Pour la variable d'induction I, une valeur convenable est assignée en sortie. Nous montrerons ensuite que cette méthode ne conduit pas à faire des hypothèses trop fortes: le non-respect de l'une quelconque conduit à un résultat qui n'est pas équivalent.

Note:

En particulier, notre méthode de démonstration n'utilise pas de propriété algébrique telle qu'associativité, commutativité... Dans le type de démarche que nous suivons, la prise en compte de ces propriétés se fait dans une première transformation de code séquentiel en code séquentiel, qui est suivi d'une transformation en code parallèle.

On obtient ainsi le :

Théorème 1

Supposons les hypothèses i) et ii) ci-dessus vérifiées. Si en outre nous avons les relations:

i) (respect des dépendances de données) (21)

pour tous $l,m,\ddot{l},\ddot{m},\overline{\sigma},\ddot{\sigma}$ tels que :

$$\Omega = A_l(m,\overline{\sigma}) \;<\; A_{\ddot{l}}(\ddot{m},\ddot{\sigma}) = \Theta$$

dans la suite Trans(Inst-1) et

$$A_l(\overline{\sigma},[1{:}N]) \geq A_{\ddot{l}}(\ddot{\sigma},[1{:}N])$$

dans la suite Trans(Inst-2) on a :

$$in(\Theta) \cap out(\Omega) = \phi$$

ii) (respect des anti-dépendances de données) (22)

pour tous $l,m,\ddot{l},\ddot{m},\overline{\sigma},\ddot{\sigma}$ tels que :

$$\Omega = A_l(m,\overline{\sigma}) \;<\; A_{\ddot{l}}(\ddot{m},\ddot{\sigma}) = \Theta$$

dans la suite Trans(Inst-1) et

$$A_l(\overline{\sigma},[1{:}N]) > A_{\ddot{l}}(\ddot{\sigma},[1{:}N])$$

dans la suite Trans(Inst-2) on a :

$$out(\Theta) \cap in(\Omega) = \phi$$

iii) (respect des dépendances en assignation) (23)

pour tous $l,m,\ddot{l},\ddot{m},\overline{\sigma},\ddot{\sigma}$ tels que :

$$\Omega = A_l(m,\overline{\sigma}) \;<\; A_{\ddot{l}}(\ddot{m},\ddot{\sigma}) = \Theta$$

dans la suite Trans(Inst-1) et

$$A_l(\overline{\sigma},[1{:}N]) > A_{\ddot{l}}(\ddot{\sigma},[1{:}N])$$

dans la suite Trans(Inst-2) on a :

$$out(\Theta) \cap out(\Omega) = \phi$$

Alors, Inst-1 est équivalente à Inst-2.

Preuve

En raisonnant par récurrence sur la longueur k de la boucle et sur la profondeur d'imbrication des boucles emboitées, il nous suffit de considérer les trois cas : ([10])

Inst-1a := (<Do> I = 1, N (A_1, A_2))

Inst-1b := (<Do> I = 1, N (B))
B := (<Do> J = 1, M (C))

Inst-1c := (<Do> I = 1, N (D))
D := X(<Index>) = <Expr>

Les hypothèses de récurrence respectives sont que A_1, A_2 et B satisfont les conclusions du Théorème. Nous notons Inst-2a, Inst-2b et Inst-2c les versions simultanées respectives.

Cas de Inst-1c:

Dans ce cas, la condition de non-interférence (12) fait qu'il suffit de s'assurer que D(i).Val = D.Val[i]. Ceci revient à s'assurer que ces deux expressions sont évaluées avec les memes données. Or, par définition, la seconde expression est évaluée dans l'etat Pred(Inst-1c),c'est à dire avec les valeurs initiales. Pour la première, D(i).Val, ceci est la conséquence immédiate de

$$j < i \Rightarrow in(D(i)) \cap out(D(j)) = \phi$$

qui est un cas particulier de l'hypothèse (21) ci-dessus

Cas de Inst-1a:

La première condition à vérifier est que:

$$A_i(m,\bar{\sigma}).Val = A_i(\bar{\sigma}).Val[m] \qquad (24)$$

c'est à dire que les résultats produits correspondent. Concernant $A_1(m,\bar{\sigma}).Val$, il suffit de s'assurer que:

$$in(A_1(\ddot{m},\ddot{\sigma})) \cap out(A_2(m,\bar{\sigma})) = \phi \qquad (25)$$

pour $m < \ddot{m}$. Ceci entraine que la présence de A_2 ne modifie pas la suite de valeurs produite par A_1 dans la boucle séquentielle. Or (25) est une conséquence de l'hypothèse (21); on applique ensuite l'hypothèse de récurrence.

Concernant $A_2(m,\bar{\sigma}).Val$, il faut maintenant s'assurer qu'il revient au meme de prendre comme données de ce calcul les valeurs finales de la boucle sur A_1. Ceci sera le cas si pour $m < \ddot{m}$:

$$out(A_1(\ddot{m},\ddot{\sigma})) \cap in(A_2(m,\bar{\sigma})) = \phi . \qquad (26)$$

Ce critère est une conséquence immédiate de l'hypothèse (22). Il revient à verifier que dans la boucle séquentielle les données utilisées par A_2 ne sont pas modifiées ultérieurement par A_1 , et que l'on est donc fondé à prendre les valeurs finales.

Nous devons maintenant prendre en compte l'assignation des valeurs ainsi produites aux

([10]) Bien que nous utilisions ci-dessous une écriture globale des instructions <DO> le procédé de reférence des occurrences du paragraphe 2 est utilisé systématiquement. En particulier on considère de facon précise toutes les occurrences de chacune des instructions à l'intérieur des boucles, et l'on ne raisonne pas sur une ocurrence globale de la boucle. La question des approximations sera traitée ci-dessous.

variables. De la meme facon que nous venons de montrer que

$$A_l(m,\bar{\sigma}).Val = A_l(\bar{\sigma}).Val[m]$$

il vient:

$$(A_l.<Index-Expr>)(m,\bar{\sigma}).Val = (A_l.<Index-Expr>)(\sigma).Val[m]$$

Donc (au sens du Lemme 1):

$$A \approx B \Rightarrow A.Var = B.Var \quad i.e. \quad out(A) = out(B) \tag{27}$$

Il nous suffit maintenant de montrer que si nous limitons les suites ordonnées introduites au Lemme 1 aux opérations sur un meme élément, les suites "quotient" sont identiques. Ceci revient à dire que l'ordre des opérations y est identique, et se déduit de la condition (23) ci-dessus et de (12).

Cas de Inst-1b:

Nous voyons que ceci peut se ramener au cas 1-a en déroulant la boucle intérieure, suivant (18) et la remarque ci-dessus. La boucle interne devient :

$$B = (I_1 ::= J=1) \ , \ C(1) \ , \ (I_2 ::= J=J+1), C(2), \cdots$$

Il n'y a pas de condition à imposer entre les I_l et les C(m), ni entre les I_l , l'ordre restant conservé. Pour les instructions C(m) les conditions sont identiques à celles d'une concaténation C_1, C_2.... et la démonstration se ramène donc au cas précédent.

3.6. Théorie de Kuck: Notions de dépendance.

3.6.1. Formulation classique.

Nous allons maintenant retrouver a partir des conditions du Théorème 1 ci-dessus les concepts liés au graphe de dépendance et la formulation usuelle de D.J.Kuck, K.Kennedy (Cf. [KKPLW80], [Ken80]). Cette formulation usuelle possède l'avantage de fournir des critères plus facilement manipulables en pratique. Contrairement à ces présentations, nous construisons ces notions comme un outil pour étudier la transformation d'un (fragment de) programme, et plus précisément représenter efficacement ses critères de validité. Sur l'exemple suivant, il est commode de discuter la différence entre notre approche et celle de K.Kennedy :

```
      DO 1 I = 1,N
        DO 2 J = 1,N
          X(J,I) = X(J, I-1) + D
2       CONTINUE
1     CONTINUE
```

Exemple 9

Dans notre approche, on peut étudier la transformation de la boucle DO 2 dans son contexte, la transformation de la boucle DO 1 etc.. A chaque transformation projetée correspond un jeu de critères d'équivalence sémantique qu'il est possible de représenter par le biais d'un graphe de dépendances. Ces graphes sont a priori distincts, meme si dans une implémentation particulière de vectoriseur ils peuvent ne pas etre disjoints.

Dans l'approche de Kennedy, cette distinction est faite par une notion de couche qui vient se superposer à un graphe de dépendance indépendant de la transformation visée. Les notions

permettant d'effectuer les distinctions nécessaires sont présentées dans Kuck [KKPLW80] par l'introduction de définitions ad hoc ("dépendance directe d'une boucle, .."). Il faut remarquer qu'une des raisons "historiques" de cette différence est que nous avons totalement séparé la phase de recherche "locale" du parallélisme que nous étudions de phases préalables permettant d'obtenir des informations globales sur le programme. Ces informations globales sont en fait à utiliser pour évaluer les diverses conditions que nous formulons sous des formes volontairement abstraites. (Cf. Jones & Muchnick eds. [JM81]). Il nous semble que les travaux sus-cités envisagent la transformation vers le parallélisme comme un prolongement d'un travail d'optimisation ce qui pousse à utiliser le meme contexte.

Le *graphe de dépendance GD* que nous allons définir représente des relations entre occurrences d'instructions (les noeuds) décrites par les figures 1,2,3,4 et 5. Les relations portent sur l'ensemble des variables utilisées ou modifiées par l'occurrence et sont représentées par des arcs orientés. Nous en déduirons divers graphes quotients plus adaptés à notre propos. Afin que ceci ait un sens nous imposerons les restrictions suivantes sur les noeuds admissibles:

i) Les instructions < Do-Conc> et < Do_Cvect> doivent etre considérées globalement: une instruction qu'ils "contiennent" ne peut etre l'objet d'une relation "isolée".

ii) Les instructions < Do-Sim> et < Do_Vect> doivent aussi etre considérées globalement mais auront pu etre distribuées en raison de la Propriété 1 ou de (13-i) respectivement.

Définition (GD):

Le Graphe de Dépendances est un graphe orienté dont les noeuds sont les occurrences d'instructions. Les arcs sont orientés et de trois types non-équivalents notés δ, δ^{-1} *et* δ^{o}. Un ou plusieurs arcs lient deux occurrences A et B si $A \overset{\sim}{<} B$, et si en outre il correspond à l'une des relations de dépendance:

Dépendance de données (Data-dependence) $A \delta B$

$$A \delta B \equiv out(A) \cap in(B) \neq \phi$$

Anti-dépendance (Anti-dependence) $A \delta^{-1} B$

$$A \delta^{-1} B \equiv out(B) \cap in(A) \neq \phi$$

Dépendance en assignation (Output-dependence) $A \delta^{o} B$

$$A \delta^{o} B \equiv out(A) \cap out(B) \neq \phi$$

Chaque arc a pour attribut le type de dépendance qu'il représente. Ces types d'arcs ne sont pas équivalents et on représente en fait 3 graphes sur un meme ensemble de noeuds.

Note:

Etant données les restrictions ci-dessus, notre relation d'ordre "précède" ($<$) est totale. Ceci n'implique pas de restriction en mode séquentiel car l'ordre est celui d'exécution, mais en impose dans les constructions parallèles. Nous pouvons essayer d'indiquer sur un exemple la raison d'etre de ce choix. Soit en effet le programme:

```
      DO-CONC  1 I ∈ (1,10)
        A
    1 CONTINUE
```

Exemple 10

Les instructions A(I) ne sont pas comparables pour notre relation d'ordre. On constate que l'existence de dépendances entre elles est contradictoire avec les hypothèses de la Définition 1 , qui donne un sens au résultat de l'exécution de ce < Do-Conc> .

En examinant la forme des relations obtenues au paragraphe précédent, il apparait immédiatement que les graphes quotients de GD par identification de familles d'occurrences sont particulièrement adaptés et plus maniables que le graphe complètement détaillé GD. Nous allons effectuer ce passage au quotient relativement à un contexte Ξ qui sera un ensemble d'instructions < Do>, < Do-Conc>, < Do-Sim>, < Do-Vect> ou < Do-Cvect> du programme.

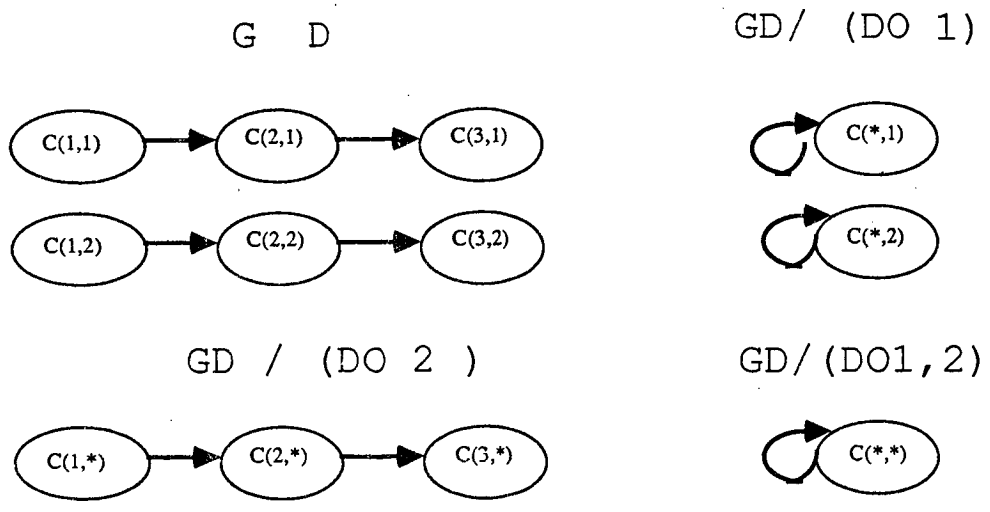
Définition (graphe quotient):

Le graphe quotient relativement à un contexte est le graphe déduit de GD par identification de toutes les occurrences indexées par les variables d'itération ou d'index des boucles de Ξ. Un arc d'un type donné lie deux noeuds (éventuellement confondus) du quotient si un arc de ce type lie dans GD deux éléments faisant respectivement partie de chacun des noeuds quotients. Ce graphe est noté GD / Ξ.

Note:

Ceci est identique au concept hiérarchique introduit par Kennedy et Allen. Ceci permet de traiter commodément des cas ou l'on s'intéresse aux boucles emboitées, avec "déroulement" ou exécution séquentielle de la boucle intérieure et pour lesquels le quotient doit etre pris uniquement relativement à la boucle externe. L'autre situation est celle ou l'on désire, à l'inverse exécuter séquentiellement la boucle externe.

Afin de fixer les idées, l'exemple 11 ci-après indique l'ensemble des graphes qu'il est possible de construire pour le programme de l'exemple 9. Nous y avons volontairement omis les arcs concernant les variables d'induction. ([11])



Exemple 11

([11]) Le traitement des variables d'induction se fait séparément, une fois qu'elles sont déterminées elles sont "remplacées par leurs valeurs" dans toutes les expressions. Ceci est finalement supporté par matériel au niveau de l'indexation des accès vectoriels et de la génération des valeurs de ces variables (Cf. par exemple Fujitsu-VP [Fuj84], [TKI85]).

Nous allons maintenant donner l'énoncé du théorème 1 ci-dessus dans ce cadre, ce qui permettra de retrouver les résultats classiques de Kuck et Kennedy. Dans tout ce qui suit Ξ est le < Do> Inst-1 : tous les noeuds correspondant à des valeurs distinctes de l'indice d'itération I sont identifiés. Tout d'abord l'hypothèse de non interférence (15) devient, en posant Ξ = (Inst-1) :

$$\textit{Il n'existe pas } A_i \textit{ avec : } A_i \delta^{o} A_i \in GD \;/\; \Xi. \tag{29}$$

Les relations (21,22,23) deviennent respectivement (30,31,32):

$$A_l(\overline{\sigma},[1{:}N]) \geq A_{\dot{l}}(\ddot{\sigma},[1{:}N]) \quad \textit{n'est possible que si l'on n'a pas:} \tag{30}$$
$$A_l(\overline{\sigma})\delta\, A_{\dot{l}}(\ddot{\sigma}) \in GD \;/\; \Xi$$

$$A_l(\overline{\sigma},[1{:}N]) > A_{\dot{l}}(\ddot{\sigma},[1{:}N]) \quad \textit{n'est possible que si l'on n'a pas:} \tag{31}$$
$$A_l(\overline{\sigma})\delta^{-1} A_{\dot{l}}(\ddot{\sigma}) \in GD \;/\; \Xi$$

ni

$$A_l(\overline{\sigma})\delta^{o} A_{\dot{l}}(\ddot{\sigma}) \in GD \;/\; \Xi \tag{32}$$

Note:

i) On note que la notion de dépendance rend implicite la condition sur l'ordre des instructions de la boucle scalaire. Une manière informelle d'exprimer la condition est que dans le graphe quotient l'arc de dépendance est orienté selon le sens d'exécution du programme.

ii) Le cas de dépendance sur le noeud lui meme est traité de facon à séparer le role de (29), propre au cas du < Do-Sim> , afin de faciliter la discussion ultérieure des cas vectoriels.

Le Théorème 1 s'écrit maintenant:

Théorème 1-bis:

Supposons Inst-1 et Inst-2 comme ci-dessus, avec la condition technique que les < Niters> des boucles incluses dans Inst-1 n'appartiennent pas à out(Inst-1). Si les conditions (29), (30), (31) et (32) ci-dessus sont vérifiées alors Inst-2 est équivalente à Inst-1.

3.6.2. Théorie de Kuck : Monotonie et approximations.

Le problème pratique est qu'il n'est pas toujours possible d'évaluer précisément les relations précédentes, que ce soit faute de savoir retrouver certaines informations sémantiques (valeur des comptes d'itération, caractère disjoint de variables, ...), ou que ce soit intentionnel dans le cas de sous programme destiné à etre exécuté pour une variété d'arguments sur lesquels on ne peut faire d'hypothèses. Il nous faut donc vérifier que les résultats ci-dessus conduisent à des relations d'équivalence exactes meme si les informations sont incomplètes. Ceci est exprimé par la monotonie : si l'on remplace dans ses relations un ensemble in() ou out () par une majoration, on obtient une condition plus restrictive:

$$(A \equiv B)_{approximatif} \Rightarrow (A \equiv B)_{exact}$$

Ceci se vérifie sur les relations (15), (21), (22), (23) par simple inspection de la forme de ces relations. L'on a donc le

Corollaire

i) Le Théorème 1 est encore vrai si dans l'évaluation des conditions (15), (21), (22) et (23) on remplace l'un des ensembles in(.) ou out(.) par un ensemble qui le contient.

ii) Ce résultat est aussi vrai si on remplace dans les memes relations l'operateur $\cap$ par $\sqcap$ de telle sorte que

$$\forall \Omega, \Theta \qquad \Omega \cap \Theta \subset \Omega \sqcap \Theta$$

L'évaluation exacte de ces relations est possible, dans la pratique, lorsque l'on peut ramener les expressions d'indice dans les tableaux à des fonctions polynomiales particulières des variables d'induction, à coefficients connus. (Cf. [Ban79], [Ken80]) Les autres facteurs qui permettent de rendre précise cette évaluation est la connaissance des invariants classiques de l'analyse globale de programmes (Cf. Cousot [Cou81], Jones & Muchnick [JM81]). Sinon, il est possible de procéder à des approximations qui se contentent du sens et de l'amplitude de variation de ces expressions d'indice. (Cf. [CRI-CFT]).

3.7. Cas des Constructions Vectorielles.

Nous allons maintenant reprendre l'étude ci-dessus dans le cas des transformations de code séquentiel vers les deux formes Vectorielles que nous avons introduites. Ceci nous donnera les critères de Vectorisation usuels avec la faculté de les transposer au cas des multipipelines grace au < Do-Cvect> .

3.7.1. Transformation d'un < Do> séquentiel en < Do-Vect> .

Ce cas se distingue du cas précédent par la suppression de la condition de non-interférence (12) et son remplacement par celle de non-récurrence (13). D'autre part, l'interprétation du < Do-Sim> est faite à partir d'une suite de transitions d'état selon la Définition 2, celle du < Do-Vect> est défini en termes d'une interprétation séquentielle (Cf. Définition 3). Les autres conditions (11) (13) n'interviennent pas directement dans la démonstration du Théorème 1 et de son équivalent ici, mais décrivent l'interprétation opérationnelle de ces constructions. Notre objet est de transformer Inst-1 en:

Inst-3 := (< Do-Vect> I=1,,N $(A_1, \ldots, A_k)$) ; I = N

Nous faisons bien entendu l'hypothèse que $A_1,,A_k$ est une suite d'assignations permises dans un < Do-Vect> . En particulier:

$$1 \leq l \leq k, 1 \leq i < j \leq N \implies out(A_l(i)) \cap in(A_l(j)) = \phi \qquad (33)$$

Le résultat est l'équivalent du théorème 1 que nous allons formuler de facon plus compacte, l'écriture des occurrences imbriquées étant sans objet:

Théorème 2

Supposons les hypothèses (33), (21), (22) et (23) vérifiées, et que N n'est pas dans out(Inst-1), alors Inst-1 est équivalente à Inst-3.

Théorème 2 (Forme simplifiée)

Supposons (33) et que N n'est pas dans out(Inst-1). Si en outre nous avons les relations:

i) (respect des dépendances de données) (34)

pour tous $l, m, \ddot{l}, \ddot{m}$ tels que :

$$\Omega = A_l(m) \quad < \quad A_{\ddot{l}}(\ddot{m}) = \Theta$$

dans la suite Trans(Inst-1) et ([12])

([12]) Nous exprimons l'ordre de deux instructions de facon explicite dans ces énoncés et non en utilisant l'indexation par l, l . Ceci prépare des versions de ces résultats ou les instructions pourraient etre réordonnés.

$$A_l([1,N]) \geq A_{\ddot{i}}([1,N])$$

dans la suite Trans(Inst-3) on a :

$$in(\Theta) \cap out(\Omega) = \phi$$

ii) (respect des anti-dépendances de données) (35)
pour tous $l, m, \ddot{l}, \ddot{m}$ tels que :

$$\Omega = A_l(m) < A_{\ddot{i}}(\ddot{m}) = \Theta$$

dans la suite Trans(Inst-1) et

$$A_l([1,N]) > A_{\ddot{i}}([1,N])$$

dans la suite Trans(Inst-3) on a :

$$out(\Theta) \cap in(\Omega) = \phi$$

iii) (respect des dépendances en assignation) (36)
pour tous $l, m, \ddot{l}, \ddot{m}$ tels que :

$$\Omega = A_l(m) < A_{\ddot{i}}(\ddot{m}) = \Theta$$

dans la suite Trans(Inst-1) et

$$A_l([1,N]) > A_{\ddot{i}}([1,N])$$

dans la suite Trans(Inst-3) on a :

$$out(\Theta) \cap out(\Omega) = \phi$$

Alors, Inst-1 est équivalente à Inst-3.

Note:

On prendra garde, si l'on utilise un argument d'équivalence par "transitivité" que Inst-2 et Inst-3 ne sont équivalentes entre elles que si l'ensemble des hypothèses des Théorèmes 1 et 2 sont vérifiées. Ceci implique que l'on a simultanément (12) et (13), ce qui en fait suffit. Nous reviendrons sur ce point ci-dessous.

Preuve:

Il suffit de reprendre la démonstration du Théorème 1, en ne considérant que les deux cas:

Inst-1a := (< Do> I = 1, N (A_1, A_2))

Inst-1c := (< Do> I = 1, N (D))
D := X(< Index>) = < Expr>

Cas de Inst-1c:

L'équivalence est ici conséquence de la définition (13) et provient de fait de la condition de Non-Récurrence (33). ([13])

Cas de Inst-1a:

([13]) Eventuellement complétée comme mentionné ci-dessus si les transferts vers la mémoire ne sont pas sérialisés.

Cette démonstration est pratiquement identique à celle qui lui correspond dans la preuve du Théorème 1. La première condition à vérifier est que:

$$A_i(m).Val = A_i.Val[m] \tag{37}$$

c'est à dire que les résultats produits correspondent. Concernant $A_1(m).Val$ il suffit de s'assurer que:

$$in(A_1(m)) \cap out(A_2(\ddot{m})) = \phi \tag{38}$$

pour $\ddot{m} < m$. Ceci entraine que la présence de A_2 ne modifie pas la suite de valeurs produite par A_1 dans la boucle séquentielle. On applique ensuite l' hypothèse de récurrence. Or (38) est une conséquence de l'hypothèse (34) .

Concernant $A_2(m).Val$ il faut maintenant s'assurer qu'il revient au meme de prendre les valeurs finales de la boucle sur A_1 . Ceci sera le cas si pour $m < \ddot{m}$:

$$out(A_1(\ddot{m})) \cap in(A_2(m)) = \phi . \tag{39}$$

Ceci résulte de l'hypothèse (35). Il revient à verifier que dans la boucle séquentielle les données utilisées par A_2 ne sont pas modifiées ultérieurement par A_1.

Nous devons maintenant prendre en compte l'assignation des valeurs ainsi produites aux variables. De la meme façon que nous venons de montrer que $A_l(m).Val = A_l.Val[m]$, il vient

$$(A_l.<Indez-Expr>)(m).Val = (A_l.<Indez-Expr>).Val[m]$$

Donc (au sens du Lemme-1):

$$A \approx B \Rightarrow A.Var \approx B.Var \Rightarrow out(A) \approx out(B) \tag{40}$$

Il nous suffit maintenant de montrer si nous limitons les suites ordonnées introduites au Lemme 1 aux opérations sur un meme élément, les suites "quotient" sont identiques. Ceci revient à dire que l'ordre des opérations concernant un meme élément y est identique, et se déduit de la condition (36) ci-dessus et de (13-ii). ([14])

3.7.2. Expression sous forme de dépendances

La Condition de Non-Récurrence (13-ii) s'exprime ici, avec $\Xi = Inst-1$:

$$Il\ n'existe\ pas\ A_i\ avec:\ A_i\ \delta\ A_i \in GD\ /\ \Xi . \tag{41}$$

Lorsque nous ne supposons pas la sérialisation des opérations avec la mémoire, la condition supplémentaire à imposer s'écrit:

$$Il\ n'existe\ pas\ A_i\ avec:\ A_i\ \delta^o\ A_i \in GD\ /\ \Xi . \tag{42}$$

Nous obtenons maintenant la forme classique du Théorème 2:

Théorème 2-bis:

Supposons Inst-1 et Inst-3 comme ci-dessus. Si les conditions (41) ([15]) , (30), (31) et (32) ci-dessus sont vérifiées alors Inst-3 est équivalente à Inst-1.

([14]) La notion d'ordre des opérations dans l'instruction <Do Vect> résulte "axiomatiquement" de (13-ii) . Si le matériel ne sérialise pas les accès mémoire, il y a lieu de modifier les hypothèses de (13) comme indiqué plus haut.

([15]) et éventuellement (42).

3.7.3. Equivalence entre < Do-Sim> , < Do-Vect> .

Nous allons montrer que si (12) (13) sont vraies simultanément alors une instruction simultanée est équivalente à la forme Vectorielle correspondante. Plus précisément nous considérons:

$$Inst\text{-}4 := (< Do\text{-}Sim > \ I \in (1,,N)\ (A_{1},,A_{k}))$$

$$Inst\text{-}5 := (< Do\text{-}Vect > \ I = 1,,N\ (A_{1},,A_{k}))$$

Nous avons alors le

Théorème 3:

Supposons que les instructions A_k soient des assignations vérifiant les hypothèses (12) et (13), alors Inst-4 est équivalente à Inst-5.

Preuve

La propriété 1 et le lemme 1 nous permettent de nous ramener immédiatement au cas k = 1. Dans le cas d'une seule instruction d'assignation, il résulte de (13-ii) que les données utilisées dans les 2 cas sont les valeurs évaluées dans l'état antérieur à l'instruction et de (12) que chacune des assignations se fait sur un élément distinct. L'équivalence en résulte.

Note:

Nous venons de prouver que lorsque k = 1, Inst-4 et Inst-5 sont équivalentes à l'instruction séquentielle < Do> associée. (Cf. (13)). Ceci n'est pas vrai dès que k = 2 et les conditions des Théorèmes 1 et 2 sont alors nécessaires.

3.7.4. Transformation d'un < Do> séquentiel en < Do-Cvect>

Notre objet est maintenant de transformer Inst-1 en Inst-6:

Inst-6 := (< Do-Cvect> $I = 1,,N\ (Synch\text{-}list)\ (A_{1},,A_{k}))$; I = N

La Définition 5 et le Théorème 3 vont nous guider dans le développement de ce cas: le < Do-Cvect> est essentiellement un < Do-Vect> dont la réalisation admet des activités concurrentes. La prise en compte de ce point est effectuée dans la Définition 5 et les hypothèses (14) moyennant quoi nous sommes, par définition, ramenés au < Do-Vect> . Il nous parait cependant souhaitable de reprendre précisément les hypothèses et de les énoncer dans les deux formalismes introduits ci-dessus. Les variables dont les manipulations sont synchronisées jouent un role particulier ici, ce qui nous conduit à poser, pour simplifier les notations:

$$\Sigma = \bigcup_{A \in < Synch\text{-}List >} A$$

Nous faisons bien entendu l'hypothèse que $A_{1},,A_{k}$ est une suite d'assignations permises dans un < Do-Cvect> :

$$1 \leq l \leq k, 1 \leq i < j \leq N \implies out(A_l(i)) \cap in(A_l(j)) = \phi \qquad (43)$$

Note:

Dans la situation ou l'écriture d'un vecteur en mémoire n'est pas sérialisée, on doit ajouter la condition :

$$1 \leq l \leq k \implies \bigcap_{1 \leq i \leq N} out(A_l(i)) = \phi \qquad (43\text{-bis})$$

Les hypothèses (21), (22) et (23) doivent bien entendu etre faites, elles interagissent avec (Def5.i et Def5.ii) pour donner:

i) (respect des dépendances de données) (44)
pour tous $l,m,\ddot{l},\ddot{m}$ tels que :

$$\Omega = A_l(m) \ < \ A_{\ddot{l}}(\ddot{m}) = \Theta$$

dans la suite Trans(Inst-1) et

$$A_l([1,N]) \geq A_{\ddot{l}}([1,N])$$

dans la suite Trans(Inst-6) on a : $in(\Theta) \cap out(\Omega) \cap \Sigma = \phi$

ii) (respect des anti-dépendances de données) (45)
pour tous $l,m,\ddot{l},\ddot{m}$ tels que :

$$\Omega = A_l(m,\bar{\sigma}) \ < \ A_{\ddot{l}}(\ddot{m}) = \Theta$$

dans la suite Trans(Inst-1) et

$$A_l([1,N]) \ > \ A_{\ddot{l}}([1,N])$$

dans la suite Trans(Inst-6) on a : $out(\Theta) \cap in(\Omega) \cap \Sigma = \phi$

iii) (respect des dépendances en assignation) (46)
pour tous $l,m,\ddot{l},\ddot{m}$ tels que :

$$\Omega = A_l(m) \ < \ A_{\ddot{l}}(\ddot{m}) = \Theta$$

dans la suite Trans(Inst-1) et

$$A_l([1,N]) \ > \ A_{\ddot{l}}([1,N])$$

dans la suite Trans(Inst-6) on a :

$$out(\Theta) \cap out(\Omega) \cap \Sigma = \phi$$

Ceci doit maintenant etre complété pour tenir compte du caractère concurrent de l'exécution par les hypothèses convenables déduites de la Definition 5:

$$i \neq j \ ==> \ \bigcup_{I=1,,N} out(A_i(I)) \ \cap \ \bigcup_{I=1,,N} out(A_j(I)) \ \subset \ \Sigma \qquad (47\text{-i})$$

$$==> \ \bigcup_{I=1,,N} out(A_i(I)) \ \cap \ \bigcup_{I=1,,N} in(A_j(I)) \ \subset \ \Sigma \qquad (47\text{-ii})$$

D'après la définition 5, et le Théorème 2, il vient:

Théorème 4:

Supposons les hypothèses (33), (44), (45), (46) et (47) vérifiées et, en outre que N n'est pas dans out(Inst-1). Alors Inst-1 est équivalente à Inst-6.

Preuve:

Ceci se déduit immédiatement du Théorème 2 et de la Définition 5.

Théorème 4-bis:

Inst-1 et Inst-6 sont équivalentes si:

i) Il n'y a pas de dépendances $\delta\ \delta^{-1}$ *ni* δ^{o} relativement aux variables qui ne sont pas dans Σ.

ii) Les hypothèses du Théorème 2-bis sont satisfaites concernant les autres dépendances.

Notre définition peut conduire à déclarer incorrectes certaines boucles dans lesquelles des dépendances existent, portant sur des variables en dehors de Σ mais qui sont correctement synchronisées par ailleurs. Ce type de difficulté parait pouvoir etre résolu par simple modification de Σ , ce qui fait que nous préférons nous en tenir à notre jeu de définitions qui conduit à privilégier le potentiel de concurrence dans la boucle, meme s'il conduit à certaines restrictions d'écriture dans le cas de synchronisations implicites. Ceci est illustré dans les figures 5 et 6 ci-dessous. Dans le premier cas, la boucle est *exclue* car il y a un *conflit apparent* sur la variable XE, alors qu'une synchronisation implicite existe via XA et XD : il suffit d'ajouter XE à Σ , sans introduire de synchronisations supplémentaires. Dans le second, la synchronisation implicite est insuffisante pour donner un sens à l'exécution concurrente: notre critère conduit bien à *interdire* cette écriture.

```
      DO-CVECT 1 I= 1, 100
      SYNCH XA,XD
          XA(I) = XB(I) + XC(I) + XE(I)
          XD(I) = XA(I) + 2.0
          XE(I) = XD(I) + XF(I)
1     CONTINUE
```

Figure 5.
Exemple de DO-Cvect non défini.

```
      DO-CVECT 1 I= 1, 100
      SYNCH XA,XD
          XA(I) = XB(I) + XC(I) + XE(I)
          XD(I) = XA(I) + 2.0
          XE(I) = XD(I) + XF(I)
          XG(I) = XE(I) ** 2
1     CONTINUE
```

Figure 6.
Exemple de DO-Cvect non défini.

3.7.5. Répartition de Boucles Séquentielles. ([17])

Nous allons maintenant montrer que les conditions nécessaires pour effectuer les transformations ci-dessus sont, pour l'essentiel identiques aux conditions permettant de répartir les boucles séquentielles. Ceci justifie l'approche la plus communément utilisée en pratique qui est de chercher des transformations de programmes séquentiels réduisant ou supprimant des arcs du graphe de dépendance, puis d'essayer de reconnaitre sur le résultat final de cette analyse celles des boucles qui peuvent etre traitées en parallèle. (Cf. [KKPL80], [KKLW80]). En particulier, cette approche ne conduit pas à faire des approximations où des hypothèses inutiles.

Elle nous semble toutefois laisser de coté certains aspects essentiels de la sémantique fine des constructions parallèles en rapport direct avec l'architecture, qui interviennent effectivement dans les réalisations les plus récentes. C'est ce dernier aspect que notre étude vise à compléter.

([17]) Nous nous limitons ici à une répartition sans réordonnancement, qui suffit ici.

(Comparer les instructions < Do-Sim>, < Do-Vect> et < Do-Cvect> d'une part, avec d'autre part l'effet produit par une boucle comportant les memes instructions machine sur un Cray-1S et un Cray-XMP. On pourra aussi s'interesser au sens d'une suite d'acces memoire eventuellement concurrents, sur les Fujitsu VP-X00.)

Par repartition de la boucle, nous entendons transformation de la boucle Inst-1 en Inst-7:

Inst-7 := $(B_1,,,B_k)$

B_l :=$(< Do>\ I=1,,N\ A_l)$

Ce type de transformation a ete etudie par divers auteurs, aussi bien en vue de la vectorisation que pour obtenir d'autres optimisations (Cf. [Abu78]). Nous ne l'etudions ici que pour preciser ses similitudes avec les precedentes transformations et nous preferons le faire dans la situation plus simple où les A_l sont de simples assignations. L'equivalent du theoreme 2 est maintenant:

Theoreme 5

Supposons les hypotheses (21), (22) et (23) verifiees et, en outre, que N ne soit pas dans out(Inst-1). Alors Inst-1 est equivalente à Inst-7.

Preuve:

Il suffit en fait de reprendre la preuve du Theoreme 2. L'hypothese (33) n'y intervient que pour traiter le cas de Inst-1c, qui est evident ici. Les autres hypotheses sont identiques et l'on effectue le meme raisonnement.

En terme de dependances, nous obtenons la forme:

Theoreme 5-bis:

Supposons Inst-1 et Inst-7 comme ci-dessus. Si les conditions (30), (31) et (32) ci-dessus sont verifiees et si N n'est pas dans out(Inst-1), alors Inst-7 est equivalente à Inst-1.

3.7.6. Stabilité et application.

On verifie immediatement que l'ensemble des criteres que nous avons produit verifient l'hypothese de monotonie et admettent donc des approximations coherentes.

Le fait que les criteres de repartition de boucles sequentielles sont un sous-ensemble de ceux necessaires pour justifier de l'application de l'une quelconque des formes vectorielles ou paralleles que nous avons introduites justifie une demarche en deux temps. Dans un premier temps, une serie de transformations de boucles sequentielles sont effectuees, visant en particulier à reduire le nombre d'arcs de dependance incompatibles avec les formes paralleles. Dans un second temps, on determine si les boucles ainsi obtenues peuvent etre transformees en boucles paralleles. A ce niveau on est conduit à differencier les divers types d'architectures. (SIMD, Vecteur,..)

4. Mise en oeuvre: Transformations et Approximations.

Nous nous interessons ici aux methodes permettant, d'une part de realiser en pratique la detection du parallelisme qui peut etre mis en oeuvre selon les criteres precedents, d'autre part de transformer le programme sequentiel pour augmenter son parallelisme potentiel. Ce programme comporte grosso-modo trois parties:

-**Analyse Semantique Globale:** les informations les plus interessantes sont celles obtenues par propagation des constantes, determination de la plage de variation des indices de boucles et de tableaux. Ces informations sont cruciales pour effectuer les tests precis de relations de dependance. Les analyses de "Liveness" de variables scalaires ou matricielles peuvent permettre l'allocation dans des registres. ([18]) La connaissance des invariants de boucle devient

([18]) Il s'agit d'un type d'optimisation particulièrement importante dans le cas des super-ordinateurs à registres vectoriels (Cray 1/XMP, Fujitsu-VP), ou possedant un niveau de memoire locale directement accessible

importante ([18]) dans le cas des tests conditionnels à effectuer en parallèle, que nous n'avons pas encore abordé.Pour le reste, l'utilisation des informations liées à l'analyse globale est classique. Nous ne reviendrons pas sur les techniques dans ce domaine pour lesquelles nous renvoyons à [JM81].

-**Détermination et Analyse des relations de Dépendance:** sur ce point, il devient crucial d'affiner au niveau des éléments de tableaux. On ne peut se satisfaire d'un test plus global, des approximations ayant toutefois été très fructueuses sur des réalisations industrielles telles que Cray Fortran (CFT Cf. [Rob], [CRI80]). La détermination des variables d'induction et leur calcul en fonction de l'indice de boucle est un préalable à cette phase. Le résultat de cette analyse est la détermination des constructions parallèles selon les critères vus ci-dessus.

-**Transformations de Programme séquentiel:** il s'agit ici de trouver des formes équivalentes dans lesquelles on a diminué ou meme supprimé les obstructions au parallélisme. Ces transformations comportent presque obligatoirement, pour etre efficaces en pratique, le renommage de variables scalaires et leur expansion vectorielle, la répartition des boucles, avec permutation des instructions. Elles sont accompagnées de transformations d'intérêt technique permettant de réduire la combinatoire telle la standardisation des limites de variation des indices de boucles.

Plutot que de donner ici un traitement systématique de ces questions, pour lesquelles une importante littérature existe, nous nous bornerons à quelques indications bibliographiques. Ceci sera *illustré* par une série d'exemples provenant d'un vectoriseur prototype "VATIL" réalisé par les auteurs et destiné à l'étude fine de ces méthodes et de leurs possibles prolongements. (Cf. A.Lichnewsky et F.Thomasset [LT,à paraitre]).

4.1. Traitement des Variables d'Induction.

On procède en deux étapes, la première consistant à normaliser la valeur initiale et le pas des variables de controle de boucle est essentiellement technique et a pour but de réduire la complexité des autres opérations. La seconde est la détection des variables d'induction et leur expression comme fonction linéaire de l'indice de boucle. La technique que nous employons dans l'exemple ci-dessous repose sur un critère global sur la boucle exprimé par des relations matricielles. (Cf. [LT,à paraitre]). Des méthodes différentes sont décrites dans [FKU75],[ACK81].

L'importance pratique de cette étape résulte de la conjugaison de plusieurs facteurs:

i) Pour déclencher les opérations vectorielles ou simultanées, il est nécessaire de connaitre les jeux d'indices de tableaux auxquelles elles vont s'appliquer. Qui plus est, les fonctions affines de l'indice de boucle seront calculées par des opérateurs matériels d'indexation implicitement ([CRI80A]) ou meme explicitement ([EE85]). La formulation explicite de ces variables est plus efficace dans ce contexte que leur calcul faisant nécessairement appel à une récurrence.

ii) Les expressions d'indice contenant ces variables vont devoir etre comparées pour la detection des dépendances lors de l'étape de parallélisation , en vertu des résultats précédents. Le fait d'avoir pu les exprimer comme fonction de la meme variable permet de les comparer efficacement. (Cf. Figures 9 et 10 ci-dessous)

iii) Les vecteurs implantés en mémoire à pas constant peuvent etre manipulés vectoriellement de facon efficace, a certaines restrictions sur le pas près. ([CRI80A], [But85]).

```
                                C (avail (1 . 0) (k . 0))
1     ;inst-101:U        >      DO 1 i = 1 ,n ,2
2     ;inst-102:S        >           k = 1+k
3     ;inst-103:S        >           j = k
4     ;inst-104:S        >           a(j) = 1+b(j)
5     ;inst-105:S        >           k = 1+k
6     ;inst-106:S        >           c(k) = a(j)+b(j)
7     ;inst-107:S        >           l = 3+1
8     ;inst-108:S        >           a(j) = c(l)
9     ;inst-101:U        >1          CONTINUE
```

Figure 7
Source avant toute transformation

```
1     ;inst-101:U        >      DO 1 #i = 1 ,1+(n-1)/2 ,1
2     ;inst-102:S        >           k = 1+k
3     ;inst-103:S        >           j = k
4     ;inst-104:S        >           a(j) = 1+b(j)
5     ;inst-105:S        >           k = 1+k
6     ;inst-106:S        >           c(k) = a(j)+b(j)
7     ;inst-107:S        >           l = 3+1
8     ;inst-108:S        >           a(j) = c(l)
9     ;inst-101:U        >1          CONTINUE
```

Figure 8
Après la standardisation des variables de controle.

```
1     ;inst-101:U        >      DO 1 #i = 1 ,1+(n-1)/2 ,1
2     ;inst-104:S        >           a(#i*2-1) = 1+b(#i*2-1)
3     ;inst-106:S        >           c(#i*2) = a(#i*2-1)+b(#i*2-1)
4     ;inst-108:S        >           a(#i*2-1) = c(#i*3)
5     ;inst-101:U        >1          CONTINUE
```

Figure 9
Après l'élimination des variables d'induction.

```
1     ;inst-116:SS       >      DOVEC 1 #i = 1 ,1+(n-1)/2 ,1
2     ;inst-104:SSS      >           a(#i*2-1) = 1+b(#i*2-1)
3     ;inst-114:SSS      >1          CONTINUE
4     ;inst-119:SS       >      DO 2 #i = 1 ,1+(n-1)/2 ,1
5     ;inst-106:SSS      >           c(#i*2) = a(#i*2-1)+b(#i*2-1)
6     ;inst-108:SSS      >           a(#i*2-1) = c(#i*3)
7     ;inst-117:SSS      >2          CONTINUE
8     ;inst-110:S        >      i = -1+(1+(n-1)/2)*2
9     ;inst-111:S        >      l = (1+(n-1)/2)*3
10    ;inst-112:S        >      j = -1+(1+(n-1)/2)*2
11    ;inst-113:S        >      k = (1+(n-1)/2)*2
```

Figure 10
Boucle (Partiellement) Vectorielle finale.

4.2. Représentation des scalaires

L'analyse du flot de données indique fréquemment que la meme variable scalaire est utilisée dans plusieurs chaines Definition --> Utilisation distinctes. Il est alors possible de *renommer* la variable pour affecter une variable différente à chacune de ces chaines. Ce faisant on supprime les dépendances artificielles créées simplement par la réutilisation de la meme variable pour ces différentes utilisations. (Cf. [KKPLW80]).

L'affectation de valeurs à une variable scalaire dans une boucle que l'on cherche à paralléliser créé automatiquement des dépendances qui empèchent la parallélisation. Qui plus est on ne

sait pas affiner l'analyse de dépendance faute de pouvoir distinguer les variables intervenant dans les diverses occurences des instructions indéxées par la variable de controle de la boucle. Il convient alors de procéder à *l'expansion vectorielle* de la variable scalaire , ce qui revient exactement à l'indexer par l'occurence de l'instruction qui lui assigne une valeur dans la boucle. Ceci est illustré ci-dessous. En pratique, des optimisations complémentaires sont possibles à la génération de code qui visent à ne pas représenter en mémoire la variable résultat de l'expansion mais à se contenter de l'utiliser comme un temporaire dans un registre vectoriel. (Cf. [KKPLW80], [CRI80]).

```
1   ;inst-101:U      >    DO 1 i = 1 ,n ,1
2   ;inst-102:S      >        cx4(i) = ar
3   ;inst-103:S      >        ar = cx5(i)
4   ;inst-104:S      >        br = ar-px5(i)
5   ;inst-105:S      >        px5(i) = ar
6   ;inst-106:S      >        cr = br-px6(i)
7   ;inst-107:S      >        px6(i) = br
8   ;inst-108:S      >        ar = br-px7(i)
9   ;inst-109:S      >        px7(i) = br
10  ;inst-101:U      >1       CONTINUE
```

Figure 11
Source avant toute transformation.

```
1   ;inst-101:U      >    DO 1 i = 1 ,n ,1
2   ;inst-102:S      >        cx4(i) = ar
3   ;inst-103:S      >        #ar1 = cx5(i)
4   ;inst-104:S      >        br = #ar1-px5(i)
5   ;inst-105:S      >        px5(i) = #ar1
6   ;inst-106:S      >        cr = br-px6(i)
7   ;inst-107:S      >        px6(i) = br
8   ;inst-108:S      >        ar = br-px7(i)
9   ;inst-109:S      >        px7(i) = br
10  ;inst-101:U      >1       CONTINUE
```

Figure 12
Après avoir renommé un scalaire.

```
1   ;inst-101:U      >    DO 1 i = 1 ,n ,1
2   ;inst-102:S      >        cx4(i) = #ar(i-1)
3   ;inst-103:S      >        ##ar1(i) = cx5(i)
4   ;inst-104:S      >        #br(i) = ##ar1(i)-px5(i)
5   ;inst-105:S      >        px5(i) = ##ar1(i)
6   ;inst-106:S      >        cr = #br(i)-px6(i)
7   ;inst-107:S      >        px6(i) = #br(i)
8   ;inst-108:S      >        #ar(i) = #br(i)-px7(i)
9   ;inst-109:S      >        px7(i) = #br(i)
10  ;inst-101:U      >1       CONTINUE
```

Figure 13
Après expansion des scalaires.

```
1   ;inst-111:S      >     ##ar1(0) = #ar1
2   ;inst-113:S      >     #ar(0) = ar
3   ;inst-115:S      >     #br(0) = br
4   ;inst-119:SS     >     DOVEC 1 i = 1 ,n ,1
5   ;inst-103:SSS    >           ##ar1(i) = cx5(i)
6   ;inst-104:SSS    >           #br(i) = ##ar1(i)-px5(i)
7   ;inst-108:SSS    >           #ar(i) = #br(i)-px7(i)
8   ;inst-117:SSS    >1          CONTINUE
9   ;inst-122:SS     >     DOVEC 2 i = 1 ,n ,1
10  ;inst-109:SSS    >           px7(i) = #br(i)
11  ;inst-106:SSS    >           cr = #br(i)-px6(i)
12  ;inst-120:SSS    >2          CONTINUE
13  ;inst-125:SS     >     DOVEC 3 i = 1 ,n ,1
14  ;inst-107:SSS    >           px6(i) = #br(i)
15  ;inst-123:SSS    >3          CONTINUE
16  ;inst-128:SS     >     DOVEC 4 i = 1 ,n ,1
17  ;inst-105:SSS    >           px5(i) = ##ar1(i)
18  ;inst-102:SSS    >           cx4(i) = #ar(i-1)
19  ;inst-126:SSS    >4          CONTINUE
20  ;inst-112:S      >     #ar1 = ##ar1(n)
21  ;inst-114:S      >     ar = #ar(n)
22  ;inst-116:S      >     br = #br(n)
```

Figure 14
Boucle vectorisée finale.

4.3. Distribution des Boucles : une Généralisation.

La comparaison des Théorèmes 1, 2, et 5 , ci-dessus suggère de procéder par une première étape de distribution de boucle en mode séquentiel, suivant le Théorème 5. Cette étape est suivie d'une étape de classification Vectorielle ou Séquentielle des boucles résultantes suivant les critères établis aux Théorèmes 1 et 2 ci-dessus. On procède enfin à la fusion des boucles séquentielles ou vectorielles lorsque cela est possible, en utilisant les critères développés aux Théorèmes 1,2,4,5.

En fait, il est possible d'aller sensiblement plus loin en réordonnant dans certains cas les instructions. Ceci permet de paralléliser des boucles qui ne satisfont pas directement les contraintes des Théorèmes 1, 2, 4 et 5. Le fondement de cette approche est une généralisation de ces résultats en remplacant respectivement Inst-2, Inst-3, Inst-6 et Inst-7 par:

$$p := \textit{permutation de } 1,\ldots,k$$

$$\textit{Inst-}2' := (<\textit{Do-Sim}> I \in (1,,N)\ (A_{p(1)},,,A_{p(k)}));\ I=N$$

$$\textit{Inst-}3' := (<\textit{Do-Vect}> I=1,,N\ (A_{p(1)},,,A_{p(k)}));\ I=N$$

$$\textit{Inst-}6' := (<\textit{Do-Cvect}> I=1,,N\ (\textit{Synch-list})\ (A_{p(1)},,,A_{p(k)}));\ I=N$$

$$\textit{Inst-}7' := (B_{p(1)},,,B_{p(k)})$$

Théorème 6:

Les résultats des Théorèmes 1, 2, 4 et 5 sont encore valables en y remplaçant Inst-2,3,6 et 7 comme indiqué ci-dessus.

Note:

Ceci ne signifie pas que l'on puisse réordonner les instructions de manière arbitraire, mais simplement que l'on peut le faire simplement après verification des relations (21)-(22)-(23) adaptées dans lesquelles la permutation figure implicitement. En effet cette permutation modifie l'ordre des occurrences d'instructions $>$.

Preuve:

Il suffit de vérifier que l'on n'a pas utilisé d'autre ordre ou séquencement dans les instructions que celui spécifié par l'ordre $>$ sur les occurrences d'instruction. En particulier on n'a pas utilisé le fait que les instructions de la forme transformée étaient identiques et dans le meme ordre que pour la boucle originale, mais simplement qu'il y avait une bijection entre elles au sens du Lemme 1.

Pour faciliter au lecteur ces transformations nous donnons ci-dessous la formulation explicite de la généralisation du Théorème 2:

Théorème 7 (Généralisation du Théorème 2)

Supposons (33) et que N n'est pas dans out(Inst-1). Si en outre nous avons les relations:

i) (respect des dépendances de données) (34)

pour tous $l,m,\ddot{l},\ddot{m}$ tels que :

$$\Omega = A_l(m) \;<\; A_{\ddot{l}}(\ddot{m}) = \Theta$$

dans la suite Trans(Inst-1) et

$$A_{p(l)}([1,N]) \;>\; A_{p(\ddot{l})}([1,N])$$

dans la suite Trans(Inst-3) on a :

$$in(\Theta) \cap out(\Omega) = \phi$$

ii) (respect des anti-dépendances de données) (35)

pour tous $l,m,\ddot{l},\ddot{m}$ tels que :

$$\Omega = A_l(m) \;<\; A_{\ddot{l}}(\ddot{m}) = \Theta$$

dans la suite Trans(Inst-1) et

$$A_{p(l)}([1,N]) \;>\; A_{p(\ddot{l})}([1,N])$$

dans la suite Trans(Inst-3) on a :

$$out(\Theta) \cap in(\Omega) = \phi$$

iii) (respect des dépendances en assignation) (36)

pour tous $l,m,\ddot{l},\ddot{m}$ tels que :

$$\Omega = A_l(m) \;<\; A_{\ddot{l}}(\ddot{m}) = \Theta$$

dans la suite Trans(Inst-1) et

$$A_{p(l)}([1,N]) \;>\; A_{p(\ddot{l})}([1,N])$$

dans la suite Trans(Inst-3) on a :

$$out(\Theta) \cap out(\Omega) = \phi$$

Alors, Inst-1 est équivalente à Inst-3'.

4.4. Distribution des boucles:

Dans la première situation ci-dessous, il n'a pas été nécessaire de réordonner les instructions.

```
1   ;inst-101:U      >    DO 1 i = 1,n,1
2   ;inst-102:S      >         x(i) = sc*y(1+i)+x(i)
3   ;inst-103:S      >         z(i) = a(i)*x(i)
4   ;inst-104:S      >         f = f+x(i)*z(i)
5   ;inst-101:U      >1        CONTINUE
```

Figure 15
Source avant transformation.

```
1   ;inst-108:S      >    DOVEC 1 i = 1,n,1
2   ;inst-102:SS     >         x(i) = sc*y(1+i)+x(i)
3   ;inst-103:SS     >         z(i) = a(i)*x(i)
4   ;inst-106:SS     >1        CONTINUE
5   ;inst-111:S      >    DO 2 i = 1,n,1
6   ;inst-104:SS     >         f = f+x(i)*z(i)
7   ;inst-109:SS     >2        CONTINUE
```

Figure 16
Boucle vectorisée finale.

Dans la situation suivante, on a réordonné les instructions et effectué le regroupement par fusion de deux boucles distribuées. Ceci est nécessaire pour parvenir à les vectoriser.

```
1   ;inst-101:U      >    DO 1 i = 1,n,2
2   ;inst-102:S      >         a(1+i) = a(i)+b(i)
3   ;inst-103:S      >         b(i) = a(i)+b(i-1)
4   ;inst-104:S      >         c(i) = b(2+i)
5   ;inst-101:U      >1        CONTINUE
```

Figure 17
Source avant transformation.

```
1   ;inst-109:SS     >    DOVEC 1 #i = 1,1+(n-1)/2,1
2   ;inst-104:SSS    >         c(#i*2-1) = b((#i*2-1)+2)
3   ;inst-102:SSS    >         a((#i*2-1)+1) = a(#i*2-1)+b(#i*2-1)
4   ;inst-107:SSS    >1        CONTINUE
5   ;inst-112:SS     >    DOVEC 2 #i = 1,1+(n-1)/2,1
6   ;inst-103:SSS    >         b(#i*2-1) = a(#i*2-1)+b((#i*2-1)-1)
7   ;inst-110:SSS    >2        CONTINUE
8   ;inst-106:S      >    i = -1+(1+(n-1)/2)*2
```

Figure 18
Boucle vectorisée finale.

4.5. Calcul des Relations de Dépendances.

Le calcul effectif des relations de dépendance apparait maintenant comme une des étapes cruciales tant dans la détection du parallélisme (Théorèmes 1,2,3,4) que dans les transformations de programmes (Théorèmes 6, 7). Du fait de la propriété de monotonie, il est possible de surestimer les ensembles in(.) et out(.) qui y interviennent, ainsi que nous l'avons montré précédemment. D'autre part il est souvent possible de vérifier que des relations de dépendances n'existent pas alors que les ensembles in(.) et out(.) sont inconnus (Cf. exemple 13 ci-après). Ces points sont essentiels en pratique, l'examen de boucles parmi les plus simples nous confrontant immédiatement avec des problèmes indécidables en toute généralité: sur l'exemple 12 ci-après, la détection des dépendances se ramène à l'évaluation de la variable J. ([19])

([19]) Ici "..." dénote l'appel à une fonction totalement générale écrite par exemple en FORTRAN.

```
      J = .....
      DO  1 I = 1,10
1     X(J-I) = X(I)
```

Exemple 12

Par contre, le cas suivant peut etre résolu sans difficulté, mais sans qu'il soit généralement possible de vérifier le programme pour un débordement d'indice de X:

```
      J = .....
      DO  1 I = 1,10
1     X(J-I)  = X(J+ I)
```

Exemple 13

Il est par contre totalement insatisfaisant de ne pas analyser les expressions d'indice, ce qui revient à introduire des dépendances entre tous les accès à une même variable. Ce point est illustré dans les figures 11-14. L'exemple 13 suggérant de plus qu'il est souhaitable de disposer d'une possibilité de calcul symbolique pour mener à bien cette opération. Les principes de base d'un test fin comparant les ensembles d'indices par des tests en arithmétique entière sont décrits de façon détaillée dans [AK82] et [Ban79]. Nous les rappelons succintement ci-après. Les aspects concernant la mise en oeuvre pratique seront détaillés dans [LT,à paraitre]. La principale difficulté d'implémentation est qu'il est souhaitable de calculer de manière symbolique certaines expressions intervenant dans les tests, avant de les simplifier et d'utiliser des propriétés arithmétiques telles que des relations de divisibilité entière. Il y a clairement un compromis généralité-efficacité à trouver, le premier objectif conduisant à la poursuite du calcul symbolique, la seconde au passage aux valeurs numériques et à l'abandon du test si elles ne sont pas disponibles. (Cf. J.Moses [Mos71])

Pour donner un exemple de mise en oeuvre de ces méthodes, supposons que nous cherchions les dépendances en assignation entre les occurrences $A(\dot{\sigma},I)$ de l'instruction A et celles $B(\dot{\sigma},I)$ de l'instruction B dans le contexte $\Xi = (DO1I{=}1,N(A,B))$. Nous supposons que

$$out(A(\dot{\sigma},I)) = A(\dot{\sigma},I).<Var> = X(a_1{*}I{+}a_0) \tag{48}$$

$$out(B(\dot{\sigma},I)) = B(\dot{\sigma},I).<Var> = X(b_1{*}I{+}b_0) \tag{49}$$

ou a_i et b_i sont constants dans la boucle en I, mais peuvent etre des expressions complexes de valeur inconnue, comme J dans l'exemple 13. Il y a dépendance si et seulement si il existe i et j avec:

$$1 \leq i,j \leq N \;\; ; \;\; a_1 i + a_0 = b_1 j + b_0 \tag{50}$$

En fait, nous allons nous contenter de déclarer qu'il n'y a pas dépendance si nous savons prouver que (50) est impossible, et qu'il y a dépendance sinon. Ceci est correct ici du fait des propriétés de monotonie vues précédemment.

4.5.1. Test de Divisibilité

Il apparait immédiatement que la condition nécessaire et suffisante pour que (50) ait des solutions $(i,j) \in \mathbf{Z}^2$ est que:

$$(b_0 - a_0) \;\; multiple \;\; de \;\; pgcd(a_1,b_1) \tag{51}$$

que l'on notera:

$$(b_0 - a_0) \in pgcd(a_1,b_1) \;.\; \mathbf{Z}$$

qui plus est une solution $(\bar{i},\bar{j})$ est fournie par application d'une variante de l'algorithme d'Euclide et l'ensemble des solutions est dépendant d'un paramètre $k \in \mathbf{Z}$ et s'écrit:

$$i = \bar{i} + \frac{k\ a_1}{pgcd(a_1,b_1)} \quad ; \quad j = \bar{j} + \frac{k\ b_1}{pgcd(a_1,b_1)} \tag{52}$$

Le **test de divisibilité** consiste à déclarer l'indépendance si (51) ne peut etre satisfaite. Le lecteur pourra estimer les étapes à réaliser de façon symbolique pour étudier l'exemple 13 et le cas suivant:

```
      J = 1 + MAX(....., 0)
      DO 1 I = 1,10
1     X(2*J*I) = X(6*I+ 1)
```

Exemple 14

4.5.2. Test de Banerjee

Dans la formulation ci-dessus on n'a pas tenu compte du fait que les indices i et j étaient compris entre 1 et N. La prise en compte de ces contraintes est dite le test de Banerjee [Ban79], il s'agit d'essayer d'affirmer l'indépendance s'il est impossible de trouver $k \in \mathbf{Z}$ avec:

$$1 \leq i = \bar{i} + \frac{k\ a_1}{pgcd(a_1,b_{1)}} \leq N \quad et \quad 1 \leq j = \bar{j} + \frac{k\ b_1}{pgcd(a_1,b_{1)}} \leq N \tag{53}$$

La mise en oeuvre pratique de ce test est décrite dans [AK82], une approximation fréquemment nécessaire lorsque N n'est pas connu est $N=\infty$. Ceci est illustré par les figures 7-10 dans le cas de variables mono-dimensionnées et par les figures 19 - 20 pour le cas des multi-indices.

```
-line-  <inst nd>  <bloc nest>     — CODE —
1       ;inst-101:U       >     DO 1 #i = 1 ,1+(n-1)/2 ,1
2       ;inst-104:S       >          a(#i*2-1,#i*2-1) = 1+b(#i*2-1)
3       ;inst-106:S       >          c(#i*2) = a(#i*2-1,#i*2-1)+b(#i*2-1)
4       ;inst-108:S       >          a(#i*2,#i*3) = c(#i*3)
5       ;inst-101:U       >1         CONTINUE
```

Figure 19

```
-line-  <inst nd>  <bloc nest>     — CODE —
1       ;inst-116:SS      >     DOVEC 1 #i = 1 ,1+(n-1)/2 ,1
2       ;inst-108:SSS     >          a(#i*2,#i*3) = c(#i*3)
3       ;inst-104:SSS     >          a(#i*2-1,#i*2-1) = 1+b(#i*2-1)
4       ;inst-114:SSS     >1         CONTINUE
5       ;inst-119:SS      >     DOVEC 2 #i = 1 ,1+(n-1)/2 ,1
6       ;inst-106:SSS     >          c(#i*2) = a(#i*2-1,#i*2-1)+b(#i*2-1)
7       ;inst-117:SSS     >2         CONTINUE
8       ;inst-110:S       >     i = -1+(1+(n-1)/2)*2
9       ;inst-111:S       >     l = (1+(n-1)/2)*3
10      ;inst-112:S       >     j = -1+(1+(n-1)/2)*2
11      ;inst-113:S       >     k = (1+(n-1)/2)*2
```

Figure 20

4.6. Approximations: l'exemple de CFT

Le fondement des approximations est la propriété de monotonie vue plus haut, et une application est de surestimer systématiquement les ensembles in(.) et out(.) chaque fois que leur évaluation n'est pas possible. En fait, nous venons de procéder par sur-estimation directe des relations de dépendances, ce qui permet des résultats précis meme lorsque les in(.) sont inconnus, comme dans l'exemple 13 et ses variantes. Pour permettre au lecteur de faire le lien

entre les résultats que nous venons d'exposer et le comportement de réalisations industrielles, nous allons discuter brièvement une approximation que fait le compilateur CFT. ([20]) [CRI80], [CRI85].

Les expressions d'indice.

Une variable d'induction < CII> est ici un entier à incrément constant en fonction de l'indice de boucle. Les expressions d'indices permettant la vectorisation doivent etre réductibles à cette forme.

Relations de dépendance approchées:

Plaçons nous dans la situation de la boucle:

```
      DO 1 I= 1,100
       X(Index-1) =    .....
       X(Index-2) =    .....
    1 CONTINUE
```

Le test effectué revient à abandonner la vectorisation si l'une des conditions suivantes n'est pas satisfaite:

1/ Index-1 et Index-2 varient dans le meme sens

2/ Si Index-1 et Index-2 croissent, Index-1 est supérieur ou égal à Index-2. Si Index-1 et Index-2 décroissent, Index-2 est supérieur ou égal à Index-1

En fait le test effectué consiste à approcher le domaine de variation d'un indice par un intervalle semi-infini dans le sens de variation de l'indice. Ceci fournit effectivement une majoration des ensembles in(.) et out(.) mentionnés ci-dessus, et permet un test extrèmement simple par rapport aux conditions de divisibilité et de Banerjee que nous avons décrites ci-dessus.

5. Extensions de la méthode.

Nous allons brièvement décrire les extensions de la méthode dans les directions du traitement des tests conditionnels, des boucles multiples. Une autre direction que nous aborderons est l'utilisation de constructions parallèles concurrentes, complétées par des méthodes de synchronisation. *Contrairement aux méthodes précédemment décrites, les auteurs ne disposent pas à ce jour d'une implémentation opérationnelle,* les exemples qui illustrerons la suite de cet exposé étant donnés à titre indicatif, et pouvant de ce fait paraitre idéalisés. Cet exposé couvre plusieurs domaines où un développement de l'outil "VATIL" est envisagé ou en cours.

5.1. Traitement des tests conditionnels.

5.1.1. Les instructions avec masque.

Nous désirons nous donner les outils pour traiter completement le langage illustré par la Figure 1. Ceci nécessite de faire entrer les tests conditionnels dans notre cadre de travail, et de le faire d'une façon qui se prète bien au parallélisme vectoriel, ou simultané. Nous allons nous guider sur l'approche des instructions avec masque des machines SIMD et vectorielles. L'idée est que l'on exécute les deux branches des tests mais qu'on ne modifie l'état ([21]) que pour celle qui est réellement suivie. Une telle instruction s'écrirait :

([20]) Le compilateur CFT décrit ici est la version 1.14, notre description est tirée de [CRI80] ou [CRI85]. Il s'agit d'un produit de Cray Research Inc.

([21]) C'est à dire qu'on ne stocke en mémoire

< Masked-Assign> := WHERE(< Logical-Expr>) :: < Var> = < Expr>

Figure 21

WHERE (L(I) .GT. I) :: A(I) = B(I)

Exemple 15

L'effet étant que pour les valeurs de i qui rendent vraie l'expression logique, B(i) est assigné à A(i). En fait, nous souhaitons nous éloigner quelque peu de cette interprétation pour introduire une interprétation fonctionnelle comprenant un opérateur de sélection. Ceci aura l'avantage de nous ramener immédiatement au cadre dans lequel nous travaillions précédemment.

D'autres formulations de cette interprétation pourront etre trouvées dans la littérature ; la méthode d'exposition de Kuck & al. passe par l'introduction d'un type de dépendance supplémentaire *" la dépendance en le controle (Control-dependence) "* que l'on réduit ensuite à une dépendance de données par le biais d'instructions masquées. (Cf. [AKPW83], [FM85]). On trouvera dans [AKPW83] un traitement détaillé des branchements généraux, alors que nous nous limitons ici au cas des < Ifs> structurés et du branchement conditionnel en sortie de boucle par < Cond-Exit> .

Pour donner un sens aux instructions < Masked-Assign> nous introduisons une fonction **when(< Cond> , < Exp1> , < Exp2>**) qui retourne le résultat de l'évaluation de < Exp1> si < Cond> =.TRUE. et celui de < Exp2> sinon

Définition

Soient < Exp1> et < Exp2> deux expressions pures ([22]) et < Cond> une expression à valeur logique pure. La fonction when retourne:

$$when(< Cond> ,< Exp1> ,< Exp2>) = \begin{array}{l} < Exp1> \; si \; < Cond> \\ < Exp2> \; sinon \end{array} \qquad (54)$$

Ceci étant nous avons l'interprétation suivante:

< Masked-Assign> := WHERE(< Logical-Expr>) :: < Var> = < Expr>

est l'équivalent de l'instruction d'assignation:

< Var> = when(< Logical-Expr> , < Expr> , < Var>)

Cette approche à l'inconvénient d'introduire < Var> dans l'ensemble in(< Masked-Assign>), et constitue une surestimation de cet ensemble, ce qui ne possède que l'inconvénient de mettre des conditions supplémentaires aux transformations. Nous discuterons de façon plus détaillée l'impact sur les critères de parallélisation. Il nous semble qu'une optimisation importante, liée à la detection des invariants, doit etre faite: lorsque dans un contexte donné, la valeur de < Logical-Expr> est constante, l'instruction doit etre remplacée soit par une assignation ordinaire, soit par une instruction nulle. ([23]) Ces optimisations ne seront pas discutées ici. Nous obtenons ainsi les ensembles:

$$in(< Masked-Assign>) = in(< Logical-Expr>) \cup in(< Expr> \cup < Var> \qquad (55)$$

$$out(< Masked-Assign>) = < Var> \qquad (56)$$

([22]) i.e. qui est evaluée sans effet de bord.
([23]) i.e. supprimée.

5.1.2. Interprétation de < If> et < Cond-Exit> .

Les techniques décrites dans [AK82], [AKPW83], [FM85] et [KKLW84] permettent de réduire ces constructions à l'application de masques adaptés aux assignations. Cette approche est radicalement opposée à celle qui est retenue dans l'étude de l'optimisation de code séquentiel ou il s'agit de minimiser les expressions calculées en générant tests et branchements. (Cf. [Set83]). Elle est directement dirigée vers l'utilisation du parallélisme architectural. (Cf. [CRI-82], [Fuj84]).

Pour décrire la réduction des tests nous introduisons l'opérateur de réduction des tests: **Red_IF** qui se définit récursivement comme suit: ([24])

Si < Do> ne contient pas < Cond-Exit>) :
Red_IF(ϕ , (< Do> I= 1,N < Bloc>)
::= (< Do> I= 1,N (Red_IF (ϕ, < Bloc>)))

Si < Do> contient un < Cond-Exit>) :
Red_IF(ϕ , (< Do> I= 1,N (< Bloc1> ; (< Cond-Exit> < Test>); < Bloc2>)))
::= $\Lambda a(1)$ = .TRUE.
(< Do> I= 1,N
Red_IF($\Lambda a(I)$,< Bloc1>) ;
$\Lambda b(I)$ = $\Lambda a(I)$ .AND. (.NOT. < Test>);
Red-IF($\Lambda b(I)$,< Bloc2>);
$\Lambda a(I+1)$ = $\Lambda b(I)$)

Dans les deux cas :
Red_IF(< *Cond*> , (< Do>))
::= Red_IF(< Cond> , Red_IF(ϕ, < Do>))

Red_IF(ϕ, (< Bloc> (A_1 ; A_2 ; ...))
::= Red_IF(ϕ , A_1) ;
Red_IF(ϕ , A_2) ; ...

Red_IF(< Cond> , (< Bloc> (A_1 ; A_2 ; ...))
::= Red_IF(< Cond> , A_1) ;
Red_IF(< Cond> , A_2) ; ...

Red_IF(ϕ , (< Assign>))
::= (< Assign>)

([24]) Nous avons noté Λx les variables créées dans l'application des règles ci-dessous. Les variables ainsi créés par deux applications d'une règle sont bien entendu distinctes.

```
Red_IF(< Cond>, (< Assign>  < Var>  < Expr>))
          ::=  (< Masked-Assign>  < Cond>  < Var>  < Expr>)


Red_IF( φ , (< If>  < Cond>  < Bloc-Then>  < Bloc-Else>))
          ::=   Red_IF(< cond>, < Bloc-Then>) ;
                Red_IF((.NOT. < Cond>) , < Bloc-Else>)

Red_IF( < Cond1>  , (< If>  < Cond2>  < Bloc-Then>  < Bloc-Else>))
          ::=   Red_IF( φ ,(< If>  (< Cond1>  .AND  < Cond2>)
                 < Bloc-Then>));
                Red_IF( φ  , (< If>  (< Cond1>  .AND. (.NOT. < Cond2>))
                 < Bloc-Else>))
```

Afin de rendre ceci plus clair, ces transformations sont illustrées dans les exemples 16, 17 et 18.

```
          DO 1 i= 1,1000
              a(i)= b(i) + c(i)
              IF (a(i) > 50 ) GOTO 2
      1       c(i)= d(i)
```

Exemple 16-a. Programme original

```
A::     #XA(1) = .TRUE.
B::    DO 1 i= 1,1000
C::       WHERE(#XA(i)) :: a(i)= b(i)+ c(i)
D::       #XB(i)= #XA(i).AND.(a(i).LE. 50)
E::       WHERE(#XB(i)) :: c(i)= d(i)
F:: 1     #XA(i+ 1) = #XB(i)
```

Exemple 16-b. Après réduction par Red_IF

```
          DO 1 i= 1,100
              a(i)= b(i)
              IF (i> 50) GOTO 2
      1       c(i)= d(i)
```

Exemple 17. Programme original

```
#XA(1) = .TRUE.
DO 1 i= 1,1000
   WHERE(#XA(i)):: a(i) = b(i)
   #XB(i)= #XA(i).AND.(i .LE. 50)
   WHERE(#XB(i)):: c(i)= d(i)
1  #XA(i+ 1) = #XB(i)
```

Exemple 17. Après réduction par Red_IF

De la meme façon que les manipulations d'indices vues plus haut conduisaient à des besoins de simplification symbolique automatique des ces expressions (Cf. [Mos71]), l'expansion des expressions logiques conduit à simplifier des expressions logiques et à la détection des sous-expressions invariantes. Les outils utilisés à ce propos par [AKPW83] sont

```
      DO 1 i=1,100
         IF (b(i) .GT. 0) THEN
         a(i)=b(i)
         ELSE
1        c(i)=d(i)
```

Exemple 18. Programme original

```
      DO 1 i=1,100
         WHERE(b(i).GT. 0):: a(i) = b(i)
         WHERE(.NOT.(b(i).GT.0)) :: c(i)=d(i)
1        CONTINUE
```

Exemple 18. Après réduction par Red_IF

dus à Quine et McCluskey ([Qui52], [McC56]); nos propres développements en cours sont basés sur [Rot60], [Tis67], et [TW82].

5.1.3. Impact sur les critères de transformation.

A priori, nous venons de ramener le problème aux méthodes précédentes, après introduction d'un *opérateur* supplémentaire : la fonction "when". Nous nous devons de vérifier cependant que la sur-estimation des relations (55) (56) ne pose pas de problèmes. La raison est que nous avons systématiquement introduit la relation :

$$< Var > \in in(< Masked-Assign >) \cap out(< Masked-Assign >) \qquad (57)$$

La première remarque est que le principe de monotonie fait que ceci ne peut introduire d'erreurs, mais risque simplement d'empecher un certain nombre de transformations. Si nous examinons maintenant cet aspect et vérifions que les résultats ne deviennent pas *systématiquement inapplicables,* en revenant sur l'ensemble des résultats ci-dessus, nous observons que:

Définition 1 (< Do-Conc>)

Cette définition n'est pas affectée, simplement < Var> peut apparaitre dans les deux relations i) et ii) au lieu de ne figurer que dans la première.

Condition de Non-Interférence (12) (< Do-Sim>)

n'est pas affectée par la sur-estimation des in(.)

Condition de Non-Recurrence (13) (< Do-Vect>)

la condition n'affecte que < Var> [i] et < Var> [j] pour $i \neq j$

Théorèmes 1, 2, 4, 5

Ces résultats ne sont pas rendus inutilisables, en effet la comparaison n'est à faire que pour $\Omega < \Theta$ avec inégalité stricte.

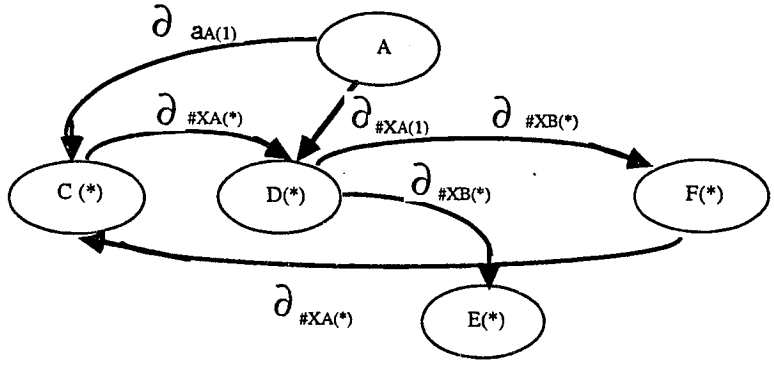
Nous donnons maintenant plusieurs exemples de l'application des résultats précédents aux formes réduites présentées en exemple ci-dessus. Il s'agit d'indications sommaires sur le comportement visé pour le produit "VATIL" qui ne possède pas ces fonctionalités à ce jour. ([25]) On notera les possibilités de Paraphrase (Kuck & al. [KKLW80-1]) et PFC (Kennedy & al.

([25]) En particulier la stratégie d'allocation de variables temporaires pour représenter les booléens, et celle de simplification des expression logique correspond à la fantaisie du rédacteur de ces exemples. Ces points seront systématisés lors de leur intégration.

[AK82]) dans ce domaine. L'idée, illustrée à l'exemple 20-c, d'isoler certaines récurrences booléennes pour lesquelles des implémentations matérielles efficaces peuvent etre construites est due à Banerjee et Gajski ([BG84]).

```
      DO-VECT 1 i= 1,100
         WHERE(b(i).GT. 0):: a(i) = b(i)
1        CONTINUE
      DO-VECT 2 i= 1,100
         WHERE(.NOT.(b(i).GT.0)) :: c(i)=d(i)
2        CONTINUE
```

Exemple 19. Vectorisation de l'exemple 18



Exemple 20-a.
Graphe de dépendance de l'exemple 16-b.

```
      #XA(1) = .TRUE.
      DO 1 i= 1,1000
         WHERE(#XA(i)):: a(i)=b(i)+c(i)
         #XB(i)=#XA(i).AND.(a(i).LE. 50)
1        #XA(i+1) =#XB(i)
      DO-VECT 2 i= 1,1000
2        WHERE(#XB(i)):: c(i)=d(i)
```

Exemple 20-b.
Vectorisation partielle de l'exemple 16.

```
      #XA(1) = .TRUE.
      DO-VECT 3 i=1,1000
         t(i)=b(i)+c(i)
         #XC(i) = (a(i) .LE. 50)
    3    #XD(i) = (t(i) .LE. 50)
      DO 1 i=1,1000
         #XB(i)=#XA(i).AND.
               ((#XA(i) .AND. #XD(i))
                .OR.
                ((.NOT.#XA(i).AND.#XC(i)))
    1    #XA(i+1) =#XB(i)
      DO-VECT 2 i=1,1000
         WHERE(#XB(i))::  c(i)=d(i)
    2    WHERE(#XA(i))::  a(i)=t(i)
```

Exemple 20-c.
Introduction de temporaires.

```
      #XA(1) = .TRUE.
      DO-VECT 3 i=1,1000
         t(i)=b(i)+c(i)
    3    #XD(i) = (t(i) .LE. 50)
      DO 1 i=1,1000
         #XB(i)=#XA(i).AND.#XD(i)
    1    #XA(i+1) =#XB(i)
      DO-VECT 2 i=1,1000
         WHERE(#XB(i))::  c(i)=d(i)
    2    WHERE(#XA(i))::  a(i)=t(i)
```

Exemple 20-d.
Après simplification de la récurrence booléenne.

On note que la seule boucle non-vectorisable contient une recurrence booléenne pour laquelle des résolutions rapides existent, utilisant l'associativité. On pourra consulter Chen et Kuck pour la résolution parallèle de recurrences plus générales, et leur implémentation matérielle. ([CK75]).

L'exemple ci-dessous montre d'une part que l'on n'aboutit pas toujours à une récurrence booléenne insoluble explicitement, et que l'analyse dépend en définitive de l'analyse des dépendances dans la boucle réduite. La première des boucles est un chargement de masque booléen et il lui correspond en général des instructions machine adéquates et parallèles ou vectorielles.

```
      DO-VECT 2 i=0,1000
         WHERE(i.LE.50) :: #XB(i)=.TRUE.
         WHERE(i.GT.50) :: #XB(i)=.FALSE.
  2   CONTINUE
      DO-VECT 1 i=1,1000
         WHERE (#XB(i-1))::a(i) = b(i)
         WHERE (#XB(i)   ::c(i) = d(i)
  1 CONTINUE
```

Exemple 21.
Vectorisation de l'exemple 17.

5.2. Vers les multi-processeurs.

L'évolution vers les multi-processeurs se fait selon plusieurs motivations. L'une est de fournir des solutions dans les domaines qui ne se prêtent pas à l'utilisation de parallélisme vectoriel ou simultané, l'autre est d'augmenter la puissance disponible sur les machines vectorielles où la possibilité d'exploiter la segmentation atteint des limites architecturales. Parmis les matériels illustrant aujourd'hui ces tendances nous trouvons d'une part les *Denelcor HEP, NYU-Ultracomputer, IBM-RP3, Caltech-Cosmic-Cube, Thinking Machines - Connection Machine,* d'autre part les *Cray-XMP/2/3, ETA-GF-10.* Le projet *Cedar* combine les deux approches et est supporté au niveau logiciel par les outils de Kuck & al. [KKPL80]. Nous montrons ici comment les techniques et considérations sémantiques illustrées dans plus haut peuvent contribuer à cette évolution. Les premiers travaux dans cette direction sont dus à Kuck et son équipe [PKL80].

5.2.1. Cray : les outils de micro-tasking. [Boo85]

Ces outils, se présentent sous la forme d'un préprocesseur qui génère des appels aux routines de synchronisation décrites dans [CRI84], *sans effectuer de controle sémantique.* Ceci offre la possibilité d'écrire des constructions basées sur le < Do-Conc> décrit plus haut suivant les deux schémas illustrés dans les exemples 22 et 23. Nous en donnons la traduction dans le cadre des constructions parallèles introduites plus haut. ([26])

```
CDIR$ DO GLOBAL
      DO 1 I=1,N
        < Bloc>
1     CONTINUE
```

Exemple 22-a
Syntaxe Cray-Microtask

```
      DO-CONC 1 I ∈ (1,...,N)
        < Bloc>
1     CONTINUE
```

Exemple 22-b

Il est important de noter que pour que cette approche soit correcte, il faut impérativement que l'utilisateur s'assure que les < Do-Conc> sont licites au sens de la Définition 1 ci-dessus. Ceci est très similaire au calcul des relations de dépendances illustré plus haut au paragraphe 4.5, est est à la portée de tous les logiciels de vectorisation automatique.

5.2.2. Synchronisations

Les idées que nous illustrons brièvement ici sont dues à Kuck (Cf. [PKL80]). Nous avons introduit plus haut la construction < Do-Cvect> pour tenir compte d'une synchronisation partielle par les données. Cette construction a son pendant sur les multiprocesseurs munis d'instructions permettant la synchronisation par les données, tels le Denelcor-HEP. L'exemple 24 montre comment ceci pourrait etre mis en oeuvre, à partir de la notion de dépendance introduite ci-dessus.

6. CONCLUSION

Nous venons de présenter des critères permettant de détecter les formes de parallélisme usuelles et de les justifier à partir d'une définition sémantique simple des constructions parallèles et vectorielles. Cette approche à l'avantage de permettre la mise en évidence simple des critères et méthodes applicables pour un modèle d'architecture et d'éxécution donnée.

([26]) Avec l'extension mineure du < Case> dont le sens est clair.

```
CDIR$ PROCESS
      <Bloc1>
CDIR$ ALSO PROCESS
      <Bloc2>
CDIR$ ALSO PROCESS
      <BlocN>
CDIR$ END PROCESS
```

Exemple 23-a
Syntaxe Cray-Microtask

```
      DO-CONC 1 A∈ (1,...,N)
         CASE A OF
            1 : <Bloc1>
            2 : <Bloc2>
            ....
            N : <BlocN>
         ESAC
1     CONTINUE
```

Exemple 23-b

Pour les architectures les plus usuelles, il suffit en fait de vérifier que l'on est dans le cadre des <Do-Conc>, <Do-Sim>, <Do-Vect> et <Do-Cvect> introduits plus haut. Nous retrouvons à partir de là les notions de dépendances de Kuck et Kennedy, ainsi que les méthodes d'analyse et de transformation qu'ils proposent.

D'un point de vue pratique, le prototype de vectoriseur VATIL, qui a été construit par les auteurs, permet de mettre en oeuvre les transformations illustrées ci dessus concretement. Il permet aussi d'expérimenter dans les diverses directions illustrées ci-dessus et va continuer d'évoluer dans la direction d'une plus grande généralité, ([27]) d'autre part pour experimenter l'impact d'architectures particulières, en suivant les voies MIMD illustrées ci-dessus et celles des unités fonctionnelles multiples supportées par le <Do-Cvect>. Les résultats déjà obtenus et les approches qu'il permet d'envisager en font un outil fondamental pour l'étude du parallélisme dans les applications et les architectures. Ce dernier point de vue parait assez largement partagé aujourd'hui (Cf. [AK82], [Boo85], [HI84], [KKLW80], [Lam74], [MK83], [YTKA], [TKI85]).

([27]) La panoplie des analyses et transformations va etre étendue en particulier aux boucles multiples.

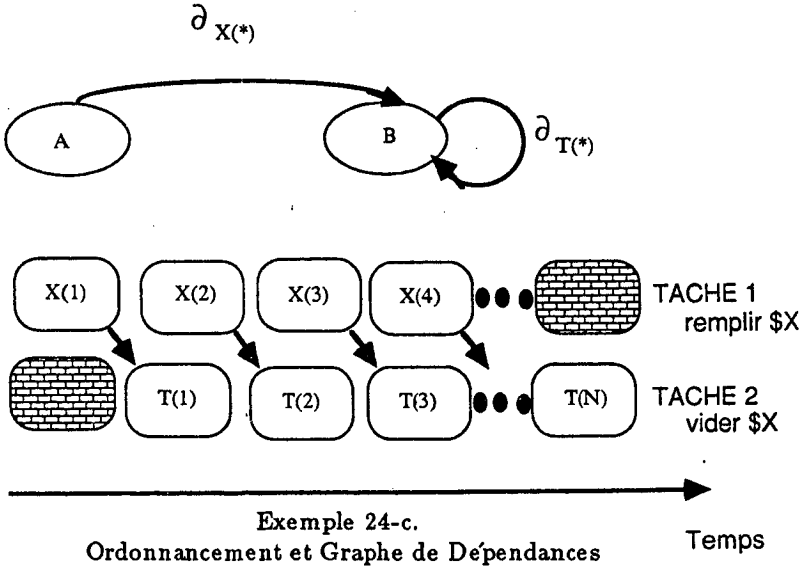
```
      DO 1 I = 1,1000
         X(I) = Y(I) + Z(I)
         T(I) = X(I) + T(I-1)
1     CONTINUE
```

Exemple 24-a. Programme initial.

```
      COMMON/MONDE/$X,Y,Z,T
      PURGE $X(*)  -- Normalement une Boucle
      PURGE $RESULT
      CREATE TASK2
      DO 1 I = 1,1000
      $X(I) = Y(I) + Z(I)
1     CONTINUE
      END

      SUBROUTINE TASK2
      COMMON/MONDE/X,Y,Z,T
      DO 2  I =1,1000
      T(I) = T(I-1) + $X(I)
2     CONTINUE
      END
```

Exemple 24-b.
Programme Parallele pour 2 Process HEP.



Exemple 24-c.
Ordonnancement et Graphe de Dépendances

BIBLIOGRAPHIE

[Abu78] ABU-SUFAH (W.)
Improving the performance of virtual memory computers
Thesis, Dept.of Computer Science,
University of Illinois at Urbana-Champaign, Nov. 1978

[AHU74] AHO (A.V.),HOPCROFT (J.E.),ULLMAN (J.D.)
The design and analysis of computer algorithms
Addison-Wesley,1974

[AK82] ALLEN (J.R.),KENNEDY (K.)
PFC: a program to convert FORTRAN to a parallel form
in [Hwa82]

[AKPW83] ALLEN (J.R.), KENNEDY (K.), PORTERFIELD (C.) & WARREN (J.),
Conversion of Control Dependence into Data Dependence
10th ACM Symp. on POPL, Austin, 1983.

[ANSI84] ANSI, Inc.
Proposal approved for Fortran 8X
X3J3/S7/Vers 89 (Working Document) , March 1984 .

[BG84] BANERJEE (U.),GAJSKI (D.D.)
Fast execution of loops with IF statements
IEEE Tr. Comp.,Vol.C-33, N0.11, Nov. 1984, pp 1030-1033

[Ban79] BANERJEE (U.)
Speedup of ordinary programs
Thesis, Dept. of Computer Science,
University of Illinois at Urbana-Champaign, Oct. 1979

[Ber66] BERNSTEIN (A.J.)
Analysis of Programs for Parallel Processing
IEEE. Trans. Computers, Vol EC-15, #5,1966.

[BG84] BANERJEE (U.),GAJSKI (D.D.)
Fast execution of loops with IF statements
IEEE Tr. Comp.,Vol.C-33, N0.11, Nov. 1984, pp 1030-1033

[Boo85] BOOTH (M.)
Multitasking - A Brief design summary
Cray Research Inc., 1985.

[But85] BUTEL (R.)
Conflicts between 2 vector transfers in CRAY-XMP computers
INRIA, Rapport de Recherche N0.148, Juin 1985

[CK75] CHEN (S.C.) , KUCK (D.J.)
Time and parallel processor bounds for linear recurrences system,
IEEE Trans. on Computers, Vol C-24, No 7, 1975.

[CH78] COUSOT (P.),HALBWACHS (N.)
Automatic Discovery of Linear Restraints among Variables in a Program
Conf. Rec. 5th ACM Symp. on POPL, Tucson, AZ, 1978.

[Cou81] COUSOT (P.)
Semantic foundations of program analysis
dans [JM81]

[CRI80] Cray Research Inc.
FORTRAN CFT Reference Manual
SR0009

[CRI80a] Cray Research Inc.
Cray 1 Computer Systems, Hardware Reference Manual
Cray Research Inc., 1980.

[CRI82] Cray Research Inc.
Cray X-MP Computer Systems, Mainframe Reference Manual
Cray Research Inc., 1982.

[CRI84] Cray Research Inc.
Multitasking User Guide
Computer Systems Technical Note, SN-0222, 1984

[CRI85] Cray Research Inc.
CFT Features Training Workbook
Cray Training Center, 1985.

[EE85] EISENBEIS (C.),ERHEL (J.)
Etude des performances du calculateur vectoriel ST100
Rapport technique INRIA, N0.51, Mai 1985

[FM85] FERRANTE (J.), MACE (M.)
On Linearizing Parallel Code
12th ACM Symp. POPL, New Orleans 1985.

[FKU75] FONG (A.), KAM (J.) & ULLMAN (J.)
Application of Lattice Algebra to Loop Optimization,
2nd ACM Symp. on POPL, Palo Alto, 1975

[Fuj84] SIEMENS / FUJITSU
Siemens System 7.800; VSP Hardware Principles of Operations
U2006-J-Z69-1-7600, Siemens A.G., Oct. 1984.

[Fuj84A] FUJITSU Ltd.
FACOM VP : General description
Fujitsu Ltd., 1984.

[GKLS83] GAJSKI (D.),KUCK (D.),LAWRIE (D.),SAMEH (A.)
Cedar - A Large Scale Multiprocessor
Proceedings of the 1983 International Conference on
Parallel Processing, pp 524-529

[Gri76] GRIES (D.)
An Exercise in Proving Parallel Programs Correct
dans "Languages Hierarchies and Interfaces",
F.L.Bauer et K.Samuelson Eds,
Lecture Notes in Comp. Sc. # 46,Springer 1976

[HI84] HORIKOSHI (H.), INAGAMI (Y.)
Japanese Supercomputers -- Overview & Perspective

[Hwa82] HWANG (Kai)
Supercomputers, design and applications
Proceedings COMPSAC'80, tutorial,1982

[JM81] JONES (N.D.), MUCHNICK (S.S.), Eds.
Program Flow Analysis
Prentice Hall, 1981.

[KKLW80] KUCK (D.J.),KUHN (R.),LEASURE (B.),WOLFE (M.)
The structure of an advanced vectorizer for pipelined processors
4th Int. Comp. Soft. & Appl. Conf., October 1980

[KKLW80-1] KUCK (D.J.),KUHN (R.),LEASURE (B.),WOLFE (M.)
The structure of an advanced retargetable vectorizer
in [Hwa82]

[KKPL80] KUCK (D.J.),KUHN (R.),PADUA (D.),LEASURE (B.),WOLFE (M.)
Dependence graphs and compiler optimizations
Proc. 8th. ACM Symp. on Principles of Programming Languages,
Williamsburg, VA, pp 207-218, Jan. 1981

[KM74] KUCK (D.J.), MURAOKA (Y.)
Bounds on the Parallel Evaluation of Arithmetic Expressions Using Associativity and Commutativity
Acta Inf., Vol 3, Fasc 3,1974

[Ken80] KENNEDY (K.)
Automatic translation of fortran programs to vector form
Rice Technical Report 476-029-4,October 1980

[Kuc81] KUCK (D.J.)
Automatic program restructuring for high-speed computation
CONPAR-81

[KLS77] KUCK (David J.),LAWRIE (Duncan H.),SAMEH(A.)
High Speed Computer and algorithms organization
Academic Press

[Lam74] LAMPORT (L.)
The parallel execution of DO-loops
CACM, February 1974, Vol.17, N0.2, pp 83-93

[Lam77] LAMPORT (L.)
The hyperplane method for an array computer
in [KLS77]

[Lam81] LAMPORT (L.)
The coordinate method for the parallel execution of iterative loops
SRI. CA-7608-0221, October 1981

[Lea76] LEASURE (Bruce Robert)
Compiling serial languages for parallel machines
University of Illinois at Urbana Champaign,
UIUCDCS-R-76-805, November 1976

[McC56] McCLUSKEY (E.J.)
Minimization of boolean functions
Bell Sys. Tech. J., 35,5,Nov.1956, pp 1417-1444.

[Mey82] MEYER (Bertrand)
Un calculateur vectoriel: le Cray-1 et sa programmation
EDF/DER/IMA, Atelier Logiciel N0.24,Janvier 1982

[Mil80] MILNER (R.)
A Calculus of Communicating Systems
Lecture Notes in Comp. Sc. # 92, Springer 1980.

[MK83] MIURA (K.), UCHIDA (K.)
FACOM Vector Processor VP-100/VP-200
Proc. NATO Advanced Research Workshop on High-Speed Computing,
Julich,1983.

[Mon85] MONGENET (C.)
Une methode de conception d'algorithmes systoliques,
resultats theoriques et realisation
These, INPL, Mai 1985

[Mos71] MOSES (J.)
Algebraic simplification: A Guide for the perplexed
2nd Symp. Symb. and Algebraic Manipulation, Comm. ACM, vol14,#8, 1971.

[NIOK] NAGASHIMA (S.), INAGAMI (Y.), ODAKA (T.), KAWABE (S.)
Design consideration for a High-speed Vector Processor:
The Hitachi S-810.

[PKL80] PADUA (D.),KUCK (D.J.),LAWRIE (D.H.)
High-speed multiprocessors and compilation techniques
IEEE Transactions on Computers, Vol.C-29, N0.9, September 1980,
pp 763-776

[Pau82] PAUL (G.)
Studies in Vector Architecture
IBM Symposium on Scientific Parallel Computing, Roma, March 1982

[Per78] PERROTT (R.)
ACTUS : A language for array and vector processors
NASA/AMES Research Center, August 1978

[PCM82] PERROTT (R.), CROOKES (D.), MILLIGAN (P.)
The programming language ACTUS
Dept.Comp.Sc., Queen's University of Belfast, November 1982

[Qui52] QUINE (W.V.)
The Problem of Simplifying truth functions
Am. Math. Monthly 59, 8, 1952, pp 521-531.

[QG84] QUINTON (P.), GACHET (P.)
Manuel d'utilisation de DIASTOL
RT INRIA-IRISA N0.41, Octobre 1984

[RL77] RAMAMOORTHY (C.V.), LI (H.F.)
Pipeline Architecture
Computing Surveys, Vol.9, N0.1, March 1977, pp 61-129

[Rob] ROBERTSON (Dale)
Introduction to Cray-1 vectorisation techniques
ECMWF, Computer Bulletin

[Rot60] ROTH (J.P.)
Minimization over booolean trees
IBM Journal, 1960, pp 543-558.

[Set83] SETHI (R.)
Control Flow aspects of Semantics-Directed Compiling
ACM Trans. Prog. Lang. and Systems, Vol 5, No.4, 1983.

[TKI85] TAMURA (H.), KAMIYA (S.), ISHIGAI (T.)
FACOM VP-100/200: Supercomputers with ease of use
Parallel Computing 2 (1985) 87-107

[Tis67] TISON (P.)
Generalisation of consensus theory and application to the minimization of boolean functions
IEEE Trans. Computers, Vol EC-16, No 4, 1967, pp 446-456.

[TW82] TEEL (B.), WILDE (D.)
A logic minimizer for VLSI PLA desigI
Proc. 19th Design Automation Conf., 1982, IEEE.

[Tow76] TOWLE (R.A.)
Control and data dependence for program transformations
University of Illinois at Urbana Champaign,
Dept. Comp. Sc., UIUCDCS-R-76-788, March 1976

[Tri85] TRIOLET (R.)
Problemes poses par l'expansion de procedures en Fortran-77
Ecole des Mines de Paris, 1985

[Wol78] WOLFE (M.)
Techniques for improving the inherent parallelism in programs
University of Illinois at Urbana Champaign,
Dept. Comp. Sc., UIUCDCS-R-78-929, July 1978

[YTKA] YASUMURA (M.), TANAKA (Y.), KANADA (Y.), AOYAMA (A.)
Compiling Algorithms and techniques
for the S-810 Vector Processor.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique